Extensión del TTF a testing de integración

Joaquín Oscar Mesuro

Director: Dr. Maximiliano Cristiá
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

8 de abril de 2015

Agradecimientos

Estoy muy agradecido de mi compañera Juanita, me invitó a intentar crecer un poco aquí.
Mis viejos, mis tíos y abuelas, que son una fuente inagotable de apoyo y aliento. Quisiera, aunque sea mínimo y de a poco, devolverles lo que significa eso para mí.
Fabri que me trajo a LCC; Ana que me recibió muy amablemente; y Maxi quien generosa y constantemente me ofrece oportunidades de trabajo y aprendizaje.
A mis amigos de año, en especial a Joa, recuerdo con mucho cariño aquel primer gesto de inclusión.

Resumen

El Test Template Framework (TTF) es un método de testing basado en modelos (Model Base Testing, MBT) para la notación Z, originalmente propuesto para el testing de unidad. En este trabajo analizamos cómo el TTF puede ser extendido para el testing de integración. Como el testing de integración está relacionado con el diseño del software, decidimos investigar la relación entre el TTF y un elemento clave de la teoría de diseño de David Parnas, la llamada relación *uses*. Proponemos cómo debería estar estructurada una especificación Z para poder aplicarle el testing de integración a través de la relación *uses*. Analizamos también el problema de la generación de *stubs*, es decir, la substitución de unidades de implementación por otras triviales y correctas en pos de facilitar el testing. El TTF ha sido automatizado aproximadamente al mismo nivel que otros MBT por la herramienta Fastest. La herramienta ha sido adaptada y modificada para poder seguir el proceso de integración propuesto. Finalmente, discutimos los tipos de errores que el testing de integración puede descubrir.

Parte del presente trabajo fue publicado como *Integration Testing in the Test Template Framework* [15] en la conferencia internacional *Fundamental Approaches to Software Engineering* (FASE) en abril de 2014.

Índice general

Capítulo 1

Introducción

Para poder realizar una descripción clara de lo que significa una extensión del TTF a testing de integración, se necesita una noción básica de los conceptos de *diseño*, *especificación*, *testing de integración*, *MBT* y *TTF*, los cuales introducimos a continuación.

El testing de software es la verificación dinámica del comportamiento del programa sobre un conjunto finito de casos de prueba. Por otro lado, los sistemas de software están construidos en base a un diseño y arquitectura que establecen partes y sus relaciones a distintos niveles de abstracción (módulos, subrutinas, subsistemas, etc). A las partes más simples se las suele denominar unidades de implementación. De acuerdo a las prácticas aceptadas en la Ingeniería de Software, después de que cada unidad de implementación ha sido testeada en forma aislada, deben ser integradas incrementalmente y testeadas [24, 36]. Esto es conocido como **testing de integración**.

Evidentemente, entonces, existe una relación entre diseño de software y testing de integración que debe ser estudiada. El **diseño** de software es definido como la descomposición de un sistema en elementos de software, la descripción de lo que cada elemento está destinado a hacer y las relaciones entre estos elementos [24]. Por ejemplo un diseño orientado a objetos (DOO) está dividido en *clases* y las relaciones pueden ser de herencia, polimorfismo, composición, etc. Dos diseños diferentes (para los mismos requerimientos) implican dos diferentes conjuntos de unidades de implementación relacionadas de dos diferentes maneras. El testing de integración, para los programas que implementan estos diseños, no puede ser el mismo porque tienen que ser consideradas las relaciones entre las unidades. Por ejemplo, un primer diseño podría tener una sola unidad, volviendo la integración trivial,

mientras que un segundo diseño podría tener muchas unidades relacionadas de maneras complejas. Por lo tanto, el estudio del testing de integración debería estar basado en una teoría del diseño de software ya que depende directamente de la estructura y de las características del mismo.

De todas las teorías o metodologías de testing este trabajo utiliza la que se denomina testing basado en modelos (MBT). El **MBT** es una técnica que permite generar casos de prueba para un programa analizando su especificación formal. Como el diseño además incluye la especificación de cada elemento de software, entonces la estructura de la especificación formal y sus relaciones con el diseño son también importantes para el testing de integración. El diseño y la especificación funcional influyen sobre los métodos de MBT porque los casos de prueba son derivados de la especificación y ejecutados en los elementos del diseño. Así, en el contexto del MBT, el testing de integración debería ser estudiado como la generación de casos de prueba guiados no solo por la especificación sino también por el diseño.

Existen numerosos métodos de MBT que varían en las notaciones formales y los algoritmos[1] para generar casos de prueba. El Test Template Framework (**TTF**) es un método de MBT propuesto para la notación Z [37, 27, 38]. Las principales actividades del TTF han sido automatizadas en la herramienta Fastest [11].

La mecánica esencial del TTF es derivar los casos de prueba desde los esquemas de operación (Sección 2.2), donde cada una de estas operaciones preferiblemente representa alguna unidad de implementación definida en el diseño (como las *clases*). Actualmente el TTF no tiene en cuenta la estructura definida por el diseño o la arquitectura para la generación de casos de prueba. Por lo tanto, el usuario del TTF se encontrará en dificultad si pretende aplicarlo para testing de integración.

Resumiendo, si por ejemplo se usa como MBT el TTF entonces la especificación es en lenguaje Z, la cual está formada por esquemas Z. Si el diseño es DOO las partes son *clases*. Al hacer testing de integración nos encontramos con dos documentos, la especificación y el diseño, es decir con esquemas Z y con clases. En ambos documentos las partes están relacionadas por composición de esquemas o por composición de instancias de clases (*objetos*), etc. De la especificación se generan los casos de prueba, del diseño las relaciones

[1] Algunos lenguajes de modelos típicos incluyen diagramas UML, notación de máquinas finitas, formalismos matemáticos como Z, B, coq. Los algoritmos dependen del modelo: probadores de teoremas, SAT-solvers, model checking, ver Sección 9.1.1.

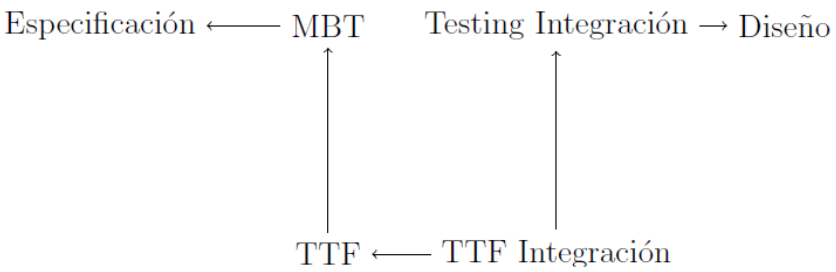

Figura 1.1: Relación entre el TTF, el MBT, testing integración, especificación y diseño.

de las clases para la integración. De esta manera, en el TTF, el estudio de las relaciones entre el diseño, la especificación, el MBT y el testing de integración encuentra un marco conceptual sólido donde realizarlo [23, 32, 16]. La Figura 1.1 representa gráficamente las relaciones básicas entre estos conceptos.

Si quisiéramos testear una parte del sistema que represente un comportamiento de relativamente alto nivel, muy probablemente signifique derivar casos de prueba de esquemas de operaciones Z que incluyan a su vez muchos otros esquemas de operaciones y de estados. Analizar automáticamente uno de estos esquemas, sin tener en cuenta los esquemas que lo forman, para generar casos de prueba no solo puede implicar un alto costo computacional sino que implica perder información valiosa sobre la estructura de la especificación. Por lo tanto, si Fastest va a ser usada en grandes sistemas es necesario extender el TTF y Fastest a testing de integración de forma tal que el testing se valga de la estructura del programa y su especificación.

Uno de los problemas del testing de integración se conoce como el problema de generación de *stubs*, el cual se trata de lo siguiente. Si las unidades de implementación están relacionadas unas con otras entonces los errores en algunas de ellas podrían causar errores en las otras. Cuando una unidad, P, depende de otra unidad, Q, testear P en soledad (testearla como unidad) es un problema porque también es necesario contar con Q. Por lo tanto, si Q tiene errores inducirá errores en P haciendo la búsqueda de la causa del error difícil y costosa. La solución aceptada es construir las llamadas unidades *stubs* las cuales imitan el comportamiento de la unidad real pero con un programa trivial y correcto.

Finalmente, tenemos como punto de partida las relaciones entre el MBT, el TTF, el testing de integración y el diseño y las necesidades teóricas y

prácticas de llevar el TTF a la integración, de allí se desprende además el problema de la generación de *stubs* para ejecutar los casos de prueba. Se propone entonces en este trabajo una manera de estructurar la especificación para reconocer las partes del diseño en la misma, un procedimiento para aliviar el problema de la generación de *stubs*, una ampliación al procedimiento del TTF para hacer efectiva la integración, y una implementación parcial de esto último en la herramienta Fastest.

La introducción a una teoría de diseño, al TTF y Fastest la presentamos en el Capítulo 2. La motivación para las cuestiones discutidas en esta tesina la damos, por medio de un ejemplo, en el Capítulo 3. En los sigientes capítulos abordamos los tres problemas principales de este trabajo: a) cómo debería estar estructurada y relacionada una especificación Z con el diseño para facilitar el testing de integración, en el Capítulo 4; b) cuál es la mejor estrategia en el TTF para integrar incrementalmente las unidades e ir obteniendo los beneficios del testing de unidad, en el Capítulo 5; y c) el problema de la generación de *stubs*, en el Capítulo 6. En el Capítulo 7 analizamos los tipos de errores que puede encontrar el TTF al ser extendido a testing de integración. En el Capítulo 8 desarrollamos un caso de estudio. Todos los resultados obtenidos en los distintos Capítulos los discutimos en el Capítulo 9.1. Por último en el Capítulo 9.1.1 establecemos comparaciones con métodos similares. Y las conclusiones en la Capítulo 10.

De aquí en más usamos al concepto de "unidad" como sinónimo de "subrutina", "función", "procedimiento" y "método". El trabajo apunta a la integración en donde el código fuente esté disponible. Todas las unidades que están integradas pertenecen al mismo ejecutable pero pueden pertenecer a diferentes módulos.

Capítulo 2

Diseño, TTF y Fastest

En este Capítulo presentamos primero una introducción a la teoría de diseño de Parnas y luego los conceptos principales del TTF y Fastest; para una presentación más profunda consultar [38, 11, 20, 12, 18].

2.1 Breve introducción a una teoría de diseño

Diseñar un sistema de software es 1) descomponer el sistema en elementos de software, 2) asignar las funciones de esos elementos y 3) establecer las relaciones entre los mismos. La misión principal es abaratar los costos del desarrollo y el mantenimiento. Se estima que el desarrollo ocupa un 33% del esfuerzo total de la producción mientras que un 66% el mantenimiento [10]. Mantener el sistema es esencialmente incorporar cambios. Finalmente, una buena teoría de diseño debe estar orientada a producir diseños que permitan reducir el costo de cambio, porque se reduce el costo de mantenimiento y es el mantenimiento el que determina el costo total del desarrollo. Los objetivos del diseño son entonces, incorporar cambios con el menor costo posible, evitando que cada cambio degrade la integridad conceptual del sistema.

Una unidad de software que implementa una funcionalidad específica se denomina *módulo*. Son fundamentales en el diseño, porque son en definitiva donde se implementan los cambios (donde está el código fuente).

Un módulo está dividido en dos secciones: interfaz e implementación. La interfaz es todo lo que está accesible a los otros módulos (ej. declaraciones de tipos, subrutinas, constantes, etc.), se dice que es su parte pública. La implementación es la forma en que la interfaz está programada, su parte pri-

vada. La interfaz de un módulo puede definirse como el conjunto de servicios que el módulo exporta [24]. Algunos de los servicios que un módulo puede exportar son: declaraciones de tipos, subrutinas, variables y constantes.

David L. Parnas [34] desarrolló una metodología de diseño para alcanzar los objetivos del diseño, llamada *diseño basado en ocultación de información*. Donde cada módulo de la descomposición se caracteriza por su conocimiento de una decisión de diseño que oculta a los demás módulos; su interfaz se elige de manera tal de revelar lo menos posible sobre su maquinaria interna. Cuando un módulo presenta una interfaz tal que los otros módulos no pueden suponer razonablemente nada sobre la implementación del módulo, se dice que la implementación está encapsulada. El objetivo pasa a ser cómo se construyen interfaces que oculten o encapsulen la implementación. Las técnicas *abstracción* y *encapsulamiento* ayudan a alcanzar el mismo. El encapsulamiento es el proceso por el cual se ocultan todos los detalles de la implementación que permanecen visibles en los servicios exportados. La abstracción consiste en lograr que la interfaz provea la menor cantidad de servicios posible y de la manera más abstracta posible. La forma básica de aplicar la abstracción al diseño de un módulo es definir la interfaz como un conjunto de subrutinas (que, por definición, son lo único que otros módulos pueden ver, usar o acceder) y luego remover todas aquellas subrutinas que pueden ser realizadas por otras.

El diseño del sistema, entonces, queda estructurado en módulos que presentan funcionalidad a través del conjunto de subrutinas que forman su interfaz. Una característica esperable del sistema es que se pueda entregar al cliente incrementalmente. Parnas [33] define los conceptos de *subconjunto útil* e *incremento mínimo* de un sistema para habilitar esta posibilidad y/o definir productos con menos o más funcionalidades orientados a diferentes clientes o hardware, etc.

Un subconjunto útil de un sistema es un conjunto mínimo de subrutinas que proveen una funcionalidad útil para el destinatario del sistema. Es mínimo porque si se quita una de las subrutinas, la funcionalidad no puede realizarse. En la actualidad a los subconjuntos útiles se los denomina versiones del sistema. Un incremento mínimo con respecto a un subconjunto útil, es un conjunto mínimo de subrutinas tal que si es agregado al subconjunto útil genera un nuevo subconjunto útil.

Parnas propone la "relación de uso" o "estructura de uso" como medio para poder calcular los subconjuntos útiles y los incrementos mínimos. Nues-

tro enfoque para la integración (Sección 5.1) está basado en esa relación.

La relación *uses* es una relación binaria entre subrutinas[1]. Si P y Q son dos subrutinas, entonces P *uses* Q si "existen situaciones en las cuales el correcto funcionamiento de P depende de la disponibilidad de una correcta implementación de Q" [33].

Parnas sugiere que las condiciones para que P *uses* Q deben ser las siguientes: P es substancialmente más simple al usar Q; Q no es substancialmente más compleja al no poder usar P; existe un subconjunto útil que contine a Q pero no a P; no existe un subconjunto útil que contiene a P pero no a Q. De estas condiciones se desprende que todos los elementos de la clausura transitiva de P deben estar en el mismo subconjunto útil en el que está P. Algo muy similar ocurre con los incrementos mínimos.

Notar que la relación *uses* difiere de la relación *calls* (o *invokes*)[2] porque: (a) si la especificación de P solo requiere que P *calls* Q entonces es suficiente para P llamar a Q cuando lo diga especificación, desde la perspectiva de P, Q puede ser correcta o no; y (b) P puede usar Q a través de compartir algunas estructuras de datos aunque el primero nunca llame al segundo. De acuerdo con Parnas, "el diseño de la jerarquía de *uses* debería ser uno de los mayores hitos en el esfuerzo de un diseño".

2.2 Test Template Framework (TTF)

El MBT está basado en la premisa de que un programa es correcto si verifica su especificación[3]. Entonces surge inmediatamente que a través de este documento se puede realizar un testing significativo. Es decir, si la especificación es lo que describe lo que debe hacer el programa, es razonable evaluar el funcionamiento de un programa a través de este documento. La especificación puede ser formal, semi-formal o informal. La técnica del TTF usa especificaciones formales (en lenguaje Z), condición necesaria si lo que se busca es automatizar el procedimiento. Concretamente el TTF permite generar casos de prueba abstractos automáticamente desde una especificación Z.

[1] Parnas usa el término programa en vez de subrutina pero otros, siguiéndolo a él usan lo segundo [5, 8].

[2] P *calls* Q si P hace una llamada (o invocación) a procedimiento de la subrutina Q.

[3] De aquí en adelante cuando se hable de especificación en realidad nos estaremos refiriendo a especificación funcional.

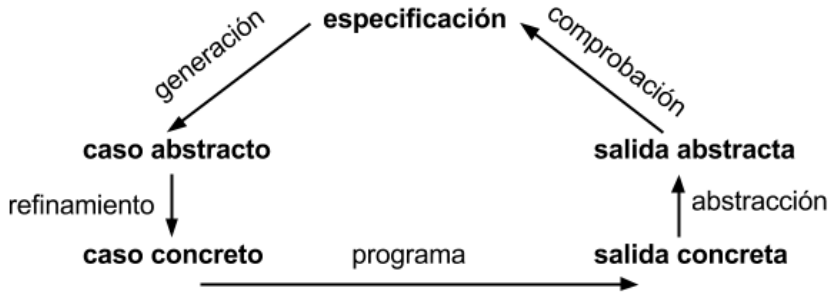

Figura 2.1: Proceso del MBT basado en especificaciones formales.

La Figura 2.1 muestra esquemáticamente el proceso entero del MBT. Conceptualmente, la especificación contiene todas las alternativas funcionales para que el programa sea correcto. Para saber si el programa es correcto, es necesario probarlo en cada una de esas alternativas. Es por eso que la generación de casos de prueba parte de la especificación. En el TTF se hace a través de las llamadas *clases de prueba*, de las cuales se derivan los *casos de prueba abstractos*, los cuales están expresados en lenguaje Z. En este informe se trata únicamente la generación de casos de prueba. Cada caso de prueba abstracto tiene que ser traducido al lenguaje de programación correspondiente para que sea suministrado, de alguna manera, al programa que se quiere probar. Ese procedimiento se llama *refinamiento* y produce los llamados *casos de prueba concretos*. El programa produce una salida por cada caso concreto, llamada *salida concreta*. Para poder comprobar la correctitud de cada salida, hay que usar la especificación, por eso es necesario traducir la salida concreta del lenguaje de programación a lenguaje Z, este proceso es la *abstracción*. La *comprobación* es simplemente, determinar si la salida abstracta corresponde al caso abstracto según la especificación.

El TTF genera casos de prueba sólo desde los esquemas de operación. El conjunto de todos los casos de prueba de un esquema de operación se llama *Valid Input Space* (*VIS*) de la operación. Consideremos el esquema de operación A con una variable de entrada $v?$ y una de estado e, y una de salida $s!$:

$$A \mathrel{\widehat=} [v? : \mathbb{Z};\ e : \mathbb{Z};\ s! : \mathbb{Z} | (e > v? \land s! = 1) \lor (e \leq v? \land s! = 0)]$$

entonces el *VIS* de A es el siguiente.

$$A^{VIS} \mathrel{\widehat=} [v? : \mathbb{Z};\ e : \mathbb{Z} | e > v? \lor e \leq v?]$$

En general el *VIS* queda definido como el conjunto de todos los valores posibles del producto cartesiano entre las variables de entrada y las de estado sujetas a las restricciones de la precondición.

$$Op^{VIS} \mathrel{\widehat{=}} [e?_1 : T_1 \ldots, e?_n : T_n, s_1 : T_{m+1} \ldots s_n : T_{m+n} | \text{pre } Op]$$

El procedimiento del TTF, para generar casos de prueba, consiste de una primera etapa donde se va particionando el *VIS*, a través de las llamadas *tácticas de prueba,* de manera que de cada partición se obtenga finalmente un elemento representante de la clase. Es decir, que se van a obtener tantos casos de prueba como partes del *VIS* se logren. El testing será más exhaustivo y fino mientras más y mejores tácticas se apliquen al *VIS*.

Una táctica de prueba es una forma sistemática de dividir el *VIS*. Algunas de las tácticas son: forma normal disyuntiva (DNF), partición estándar (SP), tipos libres (FT), etc. [11, 18]. Por ejemplo, DNF es simplemente aplicar la forma normal disyuntiva al predicado de la operación, donde cada miembro de la partición es una de las precondiciones de las disyunciones obtenidas. La táctica SP aplica a los operadores matemáticos. Cada partición estándar es una partición del dominio del operador en conjuntos llamados *sub-dominios*; cada sub-dominio está definido por las condiciones que cada operando debe cumplir (más adelante se muestra un ejemplo). FT apunta a tener en cuenta todas las formas posibles de construir un término que tiene un tipo libre. Los tipos libres en Z se utilizan tanto para definir tipos enumerados (una forma trivial de tipo inductivo) y tipos inductivos. Entonces, si en una especificación una de las variables del VIS tiene tipo libre, la táctica sugiere particionar el *VIS* en tantas clases como formas haya de construir un valor para esa variable.

Después de que una táctica de prueba es aplicada al *VIS* se obtiene una familia de clases de prueba[4]. Por ejemplo, si aplicamos DNF a A^{VIS} se producen dos clases de pruebas.

$$A_1^{DNF} \mathrel{\widehat{=}} [Op^{VIS} \mid e > v?]$$
$$A_2^{DNF} \mathrel{\widehat{=}} [Op^{VIS} \mid e \leq v?]$$

Hasta ahora podríamos obtener solo dos casos de prueba para A, uno para la clase A_1^{DNF} y otro para A_2^{DNF}. El caso genérico queda formalizado como esquemas Z de la siguiente manera:

[4]También llamadas plantillas de prueba, especificaciones de prueba, condiciones de prueba, etc.

$$Op_1^{T_1} \mathrel{\widehat{=}} [Op^{VIS} \mid P_1^{T_1}(x_1, \ldots, x_n)]$$

$$\ldots$$

$$Op_{m_1}^{T_1} \mathrel{\widehat{=}} [Op^{VIS} \mid P_{m_1}^{T_1}(x_1, \ldots, x_n)]$$

donde T_1 es el nombre de la táctica y $P_i^{T_1}(x_1, \ldots, x_n)$ para $i \in 1..m_1$ son predicados generados por T_1. Estos son los llamados predicados característicos de la clase de prueba. En otras palabras, una clase de prueba representa y define una parte del *VIS*. También se la puede ver como un conjunto de casos que satisfacen condiciones restringidas tanto por la definición del *VIS* como por la táctica.

Cabe destacar que en realidad estas clases de prueba no siempre forman una partición del *VIS*, lo cual no representa un gran problema. En ese caso se obtiene un *cubrimiento* del espacio, por ejemplo, generando más de un caso de prueba para la misma clase desde dos clases de prueba que tengan intersección no nula. Sin embargo, la mayor parte de las tácticas están diseñadas para particionar el espacio.

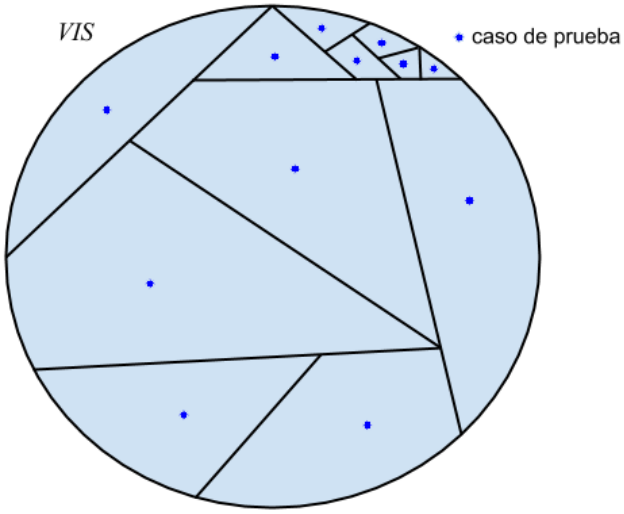

Figura 2.2: Particionamiento del VIS de un esquema de operación Z.

Luego se puede aplicar la misma táctica u otras tácticas sobre una o varias de las clases de prueba ya obtenidas (en el ejemplo, A_1^{DNF} y A_2^{DNF}). Sobre cada clase de prueba solo se debe usar una única táctica de prueba.

Tal vez, esto último sea la característica más importante del TTF, porque al aplicar otras tácticas en una o más de las clases de prueba ya generadas, se va obteniendo progresivamente condiciones de prueba más detalladas (ver Figura 2.2). Por ejemplo, si aplicamos SP al operador $\leq$ de A_2^{DNF} donde la partición propuesta para $a \leq b$ es conjugar todas las posibilidades entre $(a, <, =, >, 0)$ con $(b, <, =, >, 0)$, $(a = 0 \wedge b = 0)$, $(a < 0 \wedge b = 0)$, etc. Nos queda:

$$A_1^{SP} \widehat{=} [A_2^{DNF} \mid e < 0 \wedge v? < 0 \wedge e < v?]$$
$$A_2^{SP} \widehat{=} [A_2^{DNF} \mid e < 0 \wedge v? < 0 \wedge e = v?]$$
$$A_3^{SP} \widehat{=} [A_2^{DNF} \mid e < 0 \wedge v? = 0]$$
$$A_4^{SP} \widehat{=} [A_2^{DNF} \mid e < 0 \wedge v? > 0]$$
$$A_5^{SP} \widehat{=} [A_2^{DNF} \mid e = 0 \wedge v? = 0]$$
$$A_6^{SP} \widehat{=} [A_2^{DNF} \mid e = 0 \wedge v? > 0]$$
$$A_7^{SP} \widehat{=} [A_2^{DNF} \mid e > 0 \wedge v? > 0 \wedge e < v?]$$
$$A_8^{SP} \widehat{=} [A_2^{DNF} \mid e > 0 \wedge v? > 0 \wedge e = v?]$$

Para el caso genérico tenemos que si la táctica de prueba T_2 es aplicada a $Op_1^{T_1}$ se generan las siguientes clases de prueba:

$$Op_1^{T_2} \widehat{=} [Op_1^{T_1} \mid P_1^{T_2}(x_1, \ldots, x_n)]$$
$$\ldots$$
$$Op_{m_2}^{T_2} \widehat{=} [Op_1^{T_1} \mid P_{m_2}^{T_2}(x_1, \ldots, x_n)]$$

Notar que, si se expande $Op_1^{T_1}$ dentro de $Op_2^{T_2}$ se obtiene:

$$Op_2^{T_2} \widehat{=} [Op^{VIS} \mid P_1^{T_1}(x_1, \ldots, x_n) \wedge P_2^{T_2}(x_1, \ldots, x_n)]$$

lo que implica que un caso de prueba de $Op_2^{T_2}$ cumple ambas condiciones.

La inclusión de esquemas organiza las condiciones de prueba en el llamado árbol de prueba. En la Figura 2.3 se puede ver el árbol generado para A, el cual tiene al *VIS* como raíz, con las primeras clases de prueba en el primer nivel, y así siguiendo.

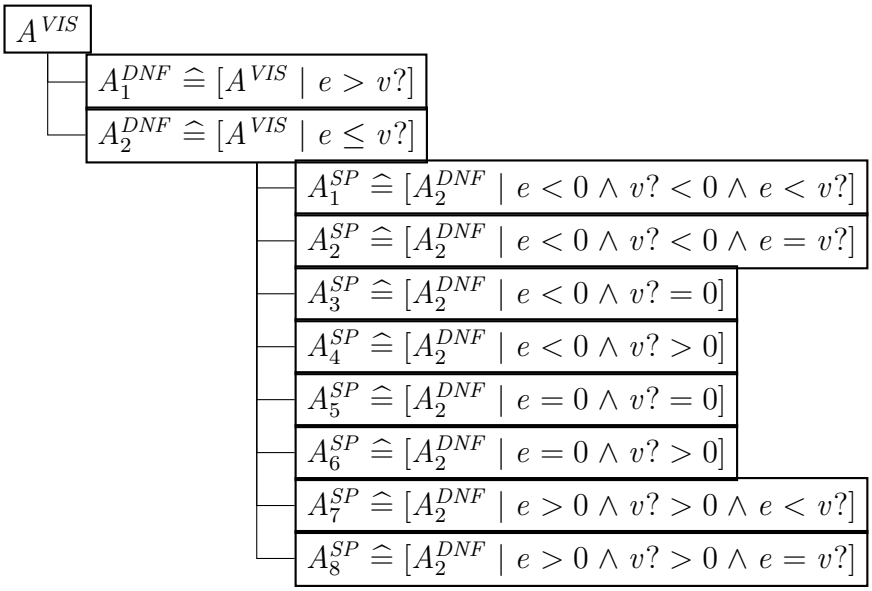

Figura 2.3: Árbol de prueba para A con dos tácticas aplicadas, DNF y SP.

Dado que en el nivel más bajo del árbol de testing (las hojas) se unen todas las condiciones de prueba de los niveles superiores, los casos son solo derivados desde las hojas. Para la operación A podríamos derivar los casos mostrados en la Figura 2.4. Con esto concluye la generación de casos de prueba en el TTF.

$$A_1^{TC} \widehat{=} [A_1^{DNF} \mid e = -2147483647 \wedge v? = -2147483648]$$
$$A_2^{TC} \widehat{=} [A_1^{SP} \mid e = -2147483647 \wedge v? = -2147483648]$$
$$A_3^{TC} \widehat{=} [A_2^{SP} \mid e = -2147483648 \wedge v? = -2147483648]$$
$$A_4^{TC} \widehat{=} [A_3^{SP} \mid e = -2147483648 \wedge v? = 0]$$
$$A_5^{TC} \widehat{=} [A_4^{SP} \mid e = -2147483648 \wedge v? = 1]$$
$$A_6^{TC} \widehat{=} [A_5^{SP} \mid e = 0 \wedge v? = 0]$$
$$A_7^{TC} \widehat{=} [A_6^{SP} \mid e = 0 \wedge v? = 1]$$
$$A_8^{TC} \widehat{=} [A_7^{SP} \mid e = 1 \wedge v? = 2]$$
$$A_9^{TC} \widehat{=} [A_8^{SP} \mid e = 1 \wedge v? = 1]$$

Figura 2.4: Casos de prueba para A generados por Fastest.

2.3 Introducción a Fastest

Fastest [9] es una herramienta que implementa la metodología de testing del TTF. La herramienta recibe una especificación Z[5] y genera de forma casi automática casos de prueba derivados de esa especificación. La versión actual de Fastest sólo genera casos de prueba abstractos[6] (casos de prueba escritos en Z) y no contempla todo el lenguaje Z aunque sí una parte significativa y muy útil. Desarrollamos brevemente el ejemplo para la operación A de la sección anterior con la herramienta para introducir los comandos básicos.

Para correr Fastest hay que pararse en el directorio de instalación e introducir el siguiente comando.

```
java -jar Fastest.jar
```

Asumiendo que la especificación está en el archivo `esp.tex` y en el mismo directorio, se puede cargar haciendo.

```
loadspec esp.tex
```

Con el comando `showloadedops` se puede ver el listado de todos los esquemas de operación que hay en la especificación. Para seleccionar una operación, se usa el comando `selop` seguido del nombre de la operación. En este caso:

```
selop A
```

Una vez seleccionada la operación es posible aplicarle las distintas tácticas de prueba. Fastest permite la posibilidad de aplicar unas diez tácticas pero siempre aplica DNF automáticamente antes de cualquier otra. Se hace efectiva la aplicación de las tácticas con el comando `genalltt` el cuál genera el árbol de prueba; el comando `showtt` lo muestra por pantalla. Entonces con:

```
genalltt
showtt
```

se obtiene por pantalla el siguiente árbol:

[5] Escrita en LaTeX y verificando el estándar ISO (Z estándar 2002).
[6] Está en etapa de prueba el refinamiento a lenguaje Java.

```
A_VIS
   !______A_DNF_1
   !______A_DNF_2
```

En el ejemplo se aplica la táctica SP al operador al operador $\leq$ de A_2^{DNF}. En Fastest se puede hacer de la siguiente manera:

```
addtactic A_DNF_2 SP \leq e \leq v?
```

Nuevamente `genalltt` para actualizar el árbol con las nuevas tácticas y `showtt` para ver el árbol.

```
A_VIS
   !______A_DNF_1
   !______A_DNF_2
         !______A_SP_1
         !______A_SP_2
         !______A_SP_3
         !______A_SP_4
         !______A_SP_5
         !______A_SP_6
         !______A_SP_7
         !______A_SP_8
```

Para poder ver cada una de las clases de prueba del árbol se utiliza el comando `showsch` seguido del nombre de la clase, por ejemplo, `showsch A_SP_1`. También se pueden ver todas las clases haciendo `showsch -tcl`. Por último, cuando se este satisfecho con el nivel de detalle que se haya obtenido con las distintas tácticas aplicadas, podemos generar los casos de prueba para cada una de ellas con el comando `genalltca`. En el ejemplo de esquema A `genalltca` genera los casos mostrados en la Figura 2.4.

Dado que el problema de la generación de casos es *indecidible*, las salidas para cada clase pueden ser tres: 1) encuentra el caso; 2) encuentra que la clase es insatisfacible, en ese caso poda automáticamente la clase; o 3) salida por *timeout*, no encuentra ningún caso y no puede determinar si la clase es insatisfacible o no. La poda es simplemente la eliminación de la clase de prueba correspondiente con todos sus descendientes. En el caso de la respuesta 3)

se puede hacer un análisis manual y si es satisfacible asignarle un caso con el comando `setfinetemodel` o en caso contrario, podarla manualmente con `prunebelow` o `prunefrom`. La herramienta cuenta además con una estrategia de poda, que resulta ser muy efectiva y extensible, con el comando `prunett` y puede ser utilizado en cualquier momento despúes de `genalltt` [12]. Lo más recomendable es usarlo después de la aplicación de cada táctica. Para obtener más detalles de cada uno de los comandos soportados por Fastest se usa el comando `help`.

Actualmente Fastest está en fase de desarrollo. Los últimos cambios corresponden al cambio del motor de generación de casos por uno basado en el lenguaje {*log*} [21]; la incorporación del proceso de refinamiento para lenguaje Java [19]; la aplicación automática de tácticas de testing [14]; y la implementación de características que permiten aplicar algunos de los conceptos de testing de integración que se presentan en este informe.

Capítulo 3

Motivación

En este Capítulo mostramos algunos de los problemas que el MBT enfrenta cuando consideramos el testing de integración. Lo hacemos a través de un ejemplo simple, en el cual se incluyen los requerimientos del sistema a implementar, dos especificaciones Z que formalizan dichos requerimientos de distinta forma y un diseño. A continuación los requerimientos.

Requerimientos generales. La funcionalidad a implementar es la extracción de un monto de dinero en una caja de ahorros en un banco por parte de un cliente titular de la caja. En el banco cada cliente se identifica por su número de documento y puede ser el titular de una caja de ahorros. Se tiene además el nombre y el domicilio de cada cliente. Las cajas de ahorros están identificadas por un único número de cuenta y guardan el monto y el historial de transacciones. El proceso además de producir la extracción de dinero, debe verificar si el usuario es un cliente, si la cuenta existe y si es el titular; si el monto que se quiere extraer es válido; y finalmente actualizar el historial de transacciones. La extracción es sólo dinero en efectivo.

Dados estos requerimientos, es razonable un diseño con dos módulos distintos, uno para Banco y otra para Caja. Ya que ambos ocultan las estructuras de datos específicas para cada entidad. Para la descripción de las interfaces de los módulos del diseño usamos el lenguaje 2MIL utilizado en Ingeniería de Software I.

Module	**Banco**
imports	Caja,Dinero,Dni,Ncta,Cliente
exportsproc	extraer(**i** NCta, **i** Dinero, **i** Dni) pedirSaldo(**i** Ncta, **i** Dni) depositar(**i** Ncta, **i** Dinero) cerrarCaja(**i** Ncta, **i** Dni) nuevoCliente(**i** Cliente)
comments	En el método extraer(), el cliente identificado por el Dni extrae de la cuenta NCta un monto de Dinero. La rutina hace uso del método extraer() de la clase Caja.

Module	**Caja**
imports	Dinero,CajaException,Historial
exportsproc	saldo():Dinero depositar(**i** Dinero) extraer(**i** Dinero) setMontoMax(**i** Dinero) getHistorial():Historial
comments	En el método extraer() se chequea si el monto ingresado es de posible extracción, si es menor al total de la caja y si es mayor a cero. Si el monto no cumple las dos condiciones se invoca la rutina CajaException::montoInvalido() y termina. Además mantiene o actualiza el historial de transacciones.

Module	**CajaException**
imports	Dinero
exportsproc	montoInvalido(**i** CajaAhorros, **i** Dinero) printStackTrace()
private	montoNegativo(**i** Dinero) montoInsuficiente(**i** Dinero)

Figura 3.1: Diseño de las interfaces con módulos 2MIL, donde Banco, Caja y CajaException son clases en un diseño orientado a objetos.

Vale aclarar que el diseño tiene más métodos de los necesarios para los requerimientos dados; están para ofrecer un poco de contexto para la inter-

pretación del ejemplo.

Presentamos a continuación una primera especificación que cumple los requerimientos.

$$DINERO == \mathbb{Z}$$
$$[NOMBRE, DOMICILIO, DNI, NUMCTA]$$
$$HIS == \operatorname{seq} \mathbb{N}$$
$$CajaAhorros == DINERO \times HIS$$
$$Cliente == NOMBRE \times DOMICILIO$$

Banco

$$cajas : NUMCTA \pfun CajaAhorros$$
$$clientes : DNI \pfun Cliente$$
$$titulares : DNI \pfun NUMCTA$$

Como puede observarse formalizamos el estado del sistema en el esquema *Banco* el cual incluye las variables *cajas*, *clientes* y *titulares*. La variable *cajas* es la relación funcional entre los números de cuenta (*NUMCTA*) y las cajas de ahorros (*CajaAhorros*). *CajaAhorros* define un par, el primer componente es de tipo *DINERO* y el segundo de tipo *HIS*, el cual es una secuencia de naturales que representa el historial de las transacciones. De la misma manera (con una función) definimos *clientes* que va de *DNI* a *Cliente*. Cada *Cliente* es un par compuesto por un *NOMBRE* y un *DOMICILIO*. Y por último, la estructura *titulares*, que relaciona los clientes con las cajas a través de los números de documentos y los números de cuentas, es decir es una función que va de *DNI* a *NUMCTA*. Tanto *clientes* como *cajas* son funciones parciales porque no todo número de documento es cliente y tiene una cuenta.

La extracción está formalizada en el esquema de operación $ExtraerOk_1$, que tiene como entrada un número de cuenta $n?$, un monto $m?$ y un dni $d?$. Se modifica la caja de ahorros correspondiente si se cumple lo siguiente: el DNI es de un cliente ($d? \in \operatorname{dom} clientes$), el número de cuenta corresponde a una cuenta válida ($n? \in \operatorname{dom} cajas$), el cliente es efectivamente titular de la cuenta ($titulares\ d? = n?$) y, por último, el monto a extraer es válido ($m? > 0$ y $m? \leq caja.1$).

$$
\begin{array}{|l}
\hline
ExtraerOk_1 \\
\Delta Banco \\
caja, caja' : CajaAhorros \\
n? : NUMCTA \\
m? : DINERO \\
d? : DNI \\
\hline
m? > 0 \\
d? \in \mathrm{dom}\ clientes \\
n? \in \mathrm{dom}\ cajas \\
titulares\ d? = n? \\
cajas\ n? = caja \\
m? \leq caja.1 \\
caja'.1 = caja.1 - m? \\
caja'.2 = caja.2 \frown \langle -m? \rangle \\
cajas' = cajas \oplus \{n? \mapsto caja'\} \\
clientes' = clientes \\
titulares' = titulares \\
\hline
\end{array}
$$

Teniendo en cuenta el diseño de arriba (Figura 3.1), resulta complicado mapear la especificación con el diseño, porque toda la extracción está representada en un solo esquema de operación. Es conveniente que en la especificación también esté reflejado el hecho de que las responsabilidades para el algoritmo de extracción estén siendo derivadas en dos subrutinas distintas.

Una gran diferencia entre la cantidad de clases del diseño y la cantidad de esquemas en la especificación podría resultar, no solo dificultosa la interpretación para la implementación, si no como veremos más adelante, la imposibilidad de obtener también un buen criterio de cubrimiento para el MBT, ya que el criterio de cubrimiento está determinado por las clases que se establecen en el diseño y el testing está guiado por los esquemas de la especificación. Por lo tanto, quien realiza la especificación debería tener en consideración asemejar las partes que componen el diseño con las que componen la especificación. Un ejemplo de especificación que se ajusta mejor al diseño podría ser el siguiente.

CajaAhorrosABanco

$\Delta Banco$
$caja, caja' : CajaAhorros$
$n? : NUMCTA$
$d? : DNI$

$d? \in \text{dom } clientes$
$n? \in \text{dom } cajas$
$titulares\ d? = n?$
$cajas\ n? = caja$
$cajas' = cajas \oplus \{n? \mapsto caja'\}$
$clientes' = clientes$
$titulares' = titulares$

CAExtraerOk

$caja, caja' : CajaAhorros$
$m? : DINERO$

$m? > 0$
$m? \leq caja.1$
$caja'.1 = caja.1 - m?$
$caja'.2 = caja.2 \frown \langle -m? \rangle$

$$ExtraerOk == CajaAhorrosABanco \land CAExtraerOk$$

En esta segunda especificación[1], la operación de extracción está representada por dos esquemas de operaciones donde se ve claramente que las responsabilidades del método Banco :: extraer para la extracción están en el primer esquema y las de Caja :: extraer en el segundo. De esta manera, al aplicar el TTF se prueban separadamente las subrutinas de extracción correspondientes a la modificación de una caja y las subrutinas encargadas de hacer lo propio a nivel del banco. En este caso el mapeo es directo como puede verse en la Figura 3.2

[1]Notar que tanto para *CajaAhorrosABancoOk* como para *CAExtraerOk* no están especificados los esquemas de error correspondientes, como si lo están más adelante, en un ejemplo similar (no idéntico) más completo, en la Sección 8.

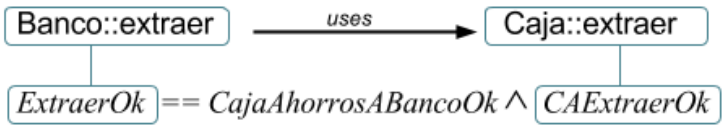

Figura 3.2: Relación diseño-especificación para la segunda especificación.

En resumen, tenemos dos especificaciones para los mismos requerimientos. Dado que hay dos especificaciones diferentes para el diseño, los casos generados por un método de MBT deberían ser diferentes cuando se aplican a cada especificación. Más aún, la especificación establece que la correctitud de Banco :: extraer depende de la correctitud de Caja :: extraer, cuestión que está mejor expresada en la segunda especificación, ya que *CAExtraerOk* es la parte de *ExtraerOk* que atiende a la funcionalidad de una caja y *CajaAhorrosABanco* hace lo propio con el banco.

Esta dependencia debería imponer un orden para testear las unidades que convendría ser tenido en cuenta por el método de MBT. Si por el contrario, Banco :: extraer es testeado antes que Caja :: extraer haya pasado todos sus tests, los errores en Banco :: extraer podrían ser difíciles de detectar porque podrían venir desde la misma clase o desde Caja :: extraer. Como sea, incluso si Caja :: extraer ha pasado todos sus tests, el error no se podría adjudicar sólo a Banco :: extraer, porque no ha sido demostrada la correctitud de la primera, sólo fue testeada. Entonces, los errores encontrados podrán adjudicarse solo a Banco :: extraer cuando el testing se haya hecho con un *stub* (correcto) para Caja :: extraer, o se haya demostrado que los casos de prueba cuando corren en Caja :: extraer se invocan idénticamente que cuando este método fue testeado. Y como Caja :: extraer ha pasado todos sus tests solo queda culpar a Banco :: extraer.

Finalmente, si, por ejemplo, Caja :: extraer llama a una rutina de error, err, cuando $m? \leq 0$, ¿se debería testear err antes de Banco :: extraer? No sería necesario porque err no es parte de la especificación de Banco :: extraer (*ExtraerOk*). En otras palabras, la correctitud de Banco :: extraer no depende de err. Esto implica que cualquier *stub* de err servirá durante el testing de Banco :: extraer. De esta manera el *stub* se puede construir automáticamente a partir de la información de diseño. A continuación mostramos dos ejemplos de un *stub* para err (en el ejemplo, err es CajaException :: montoInvalido) en Java:

```
public class CajaException{
        public void montoInvalido(CajaAhorros c, Dinero d){
                return;
        }
}
public class CajaException{
        public void montoInvalido(CajaAhorros c, Dinero d){
                this.printStackTrace();
                return;
        }
}
```

El primero no hace nada, y el segundo solo imprime la traza de llamadas. Son dos *stubs* que carecen de error alguno y susceptibles de ser generados automáticamente por su simpleza únicamente con la información de diseño.

En conclusión, una adaptación conveniente de un método MBT puede ayudar de varias maneras durante el testing de integración como mostramos en el resto del informe.

Capítulo 4

Estructuración de una especificación Z para Testing de Integración

En este Capítulo tratamos las siguientes cuestiones: usar Z para especificar subrutinas que luego van a ser integradas (Sección 4.1); mantener el espacio de las operaciones Z tan chico como sea posible (Sección 4.2); y la representación de la relación *uses* de Parnas en Z (Sección 4.3).

4.1 Representación de unidades de implementación

Los esquemas Z pueden ser usados para muchos propósitos, desde la definición de tipos de datos hasta la codificación de las propiedades de comportamiento del modelo. El lenguaje Z es flexible en cuanto que ofrece la posibilidad de especificar comportamiento con distintos elementos (o combinación de ellos), como con las definiciones axiomáticas, esquemas de estado y de operación y definiciones genéricas [37, 27]. En este trabajo sólo consideramos que el comportamiento de la implementación es especificado por medio de esquemas. Pero no se pierde nada al restringir el trabajo solo a esquemas. La restricción es en parte por el TTF y por su implementación en Fastest[1]. Casi la totalidad de estas restricciones pueden verse como las convenciones de codificación para los lenguajes de programación.

[1] Otras restricciones son necesarias si se quiere usar Fastest de la manera más efectiva [9].

Un **esquema es de operación** si declara, luego de una expansión total, como mínimo una variable de entrada o de estado, y como mínimo una de salida o variable de estado primada. El TTF debe generar casos de prueba solo para aquellos esquemas de operaciones que correspondan a subrutinas. Formalmente, se deben generar casos de prueba para el esquema A si y solo si es la especificación de una subrutina P. Por ejemplo, en relación a los esquemas definidos en el Capítulo 3, se deberán generar casos para *ExtraerOk* y *CAExtraerOk* pero no para *CajaAhorrosABanco*, por no corresponder a una subrutina del diseño. Los casos que cubren la funcionalidad especificada en *CajaAhorrosABanco* serán generados cuando sea analizado *ExtraerOk* ya que el primero será expandido dentro de último.

Z permite formar expresiones entre esquemas tanto con los operadores de esquemas como con la inclusión de esquemas, lo que implica que el mapeo entre esquemas y subrutinas puede hacerse con esquemas que estén compuestos por varios de los mismos. Por ejemplo, el esquema de operación A puede definirse $A == \mathcal{SE}(A_1, \ldots, A_n)$ donde $\mathcal{SE}$ es cualquier expresión de esquemas dependiendo de los esquemas A_i para $i \in 1 \,..\, n$. Entonces, si hay una subrutina P, cuya especificación es A, los usuarios deberán generar casos de prueba desde A para P. Si hay una subrutina R, cuya especificación es A_i, para algún $i \in 1 \,..\, n$, entonces se deberán generar casos también desde A_i para testear R (por ejemplo *CAExtraerOk*). Pero si no hay subrutina cuya especificación sea A_i entonces no se deberán generar casos de prueba desde este esquema (por ejemplo *CajaAhorrosABanco*). Cuando P invoca otras subrutinas entonces A (o la expresión de esquema que la define) debe ajustarse a algunas reglas simples dadas en la sección. 4.3.

4.2 Reducción del espacio de estados

Si los esquemas Z son usados para especificar subrutinas, entonces los esquemas deben declarar solo las variables necesarias para especificar la subrutina correspondiente. Si se declaran variables que no son (estrictamente) necesarias, el caso de prueba se vuelve innecesariamente complejo [19]. Esto es particularmente importante para las variables de estado. En efecto, una de las primeras tareas cuando se escribe una especificación Z es definir su espacio de estados a través de esquemas de estados. Un **esquema de estados** es un esquema que declara algunas variables de estado y usualmente incluye algunos invariantes de estado. Luego, este esquema es incluido en la forma de $\Delta Schema$ o $\Xi Schema$ en cada operación definida en la especificación. De esta manera, cada operación puede acceder a todas las variables de estado.

Por lo tanto, nuestra propuesta es definir esquemas de estados con la menor cantidad de variables posibles; el mejor caso sería que cada operación tuviera su propio esquema de estado declarando solo aquellas variables de estado que la operación necesita. Si algunas operaciones necesitan exactamente el mismo conjunto de variables de estados, entonces el mismo esquema de estado puede ser incluido en todas ellas. Esta recomendación tiene impacto en el costo testing de unidad cuando luego progresivamente se hace el testing de integración dado que el espacio de estados se va conservando lo más reducido posible [6].

Escribir los invariantes de estado en un esquema separado (y no dentro del esquema de estado como usualmente se hace) también ayuda en la simplificación de la generación de casos de prueba, como ya se ha mostrado [11]. Como sea, no tiene un impacto especial al pasar del testing de unidad al de integración. Si se usa el invariante por separado, el predicado del invariante aparece solo en la *obligación de prueba*, que es un teorema que el modelo debe cumplir para certificar que cumple el invariante correspondiente [13]. Entonces al no estar los predicados de los invariantes, en los esquemas de operación, no tienen ningún efecto en la generación de casos de prueba.

Por otro lado, si se incluyen en el esquema de estado, entonces aparecen como precondición en todas los esquemas de operación en que el esquema de estado esté incluido. El impacto de esta precondición extra, afecta de la misma manera cuando se pasa al testing de integración que como lo hace en el testing de unidad. Esto es así por la estrategia de testing de integración establecida en este informe en la Sección 5.2.

La estrategia que definimos allí consiste, básicamente, en sacar de los esquemas compuestos, los esquemas incluidos ya probados con todas las variables únicamente definidas en el último. Dicho de otro modo, si tenemos un esquema de operación compuesto por otros esquemas ya probados, tengan invariantes o no, no van a afectar la generación de casos de prueba, porque directamente son sacados del esquema compuesto. Por otro lado, si el invariante está definido para las variables del esquema compuesto, entonces el predicado del invariante afecta la generación de casos de prueba, pero lo hace de la misma manera que durante el testing de unidad. Es decir, si un esquema tiene un invariante en su predicado, sólo afectará a ese esquema durante el testing, y no a los esquemas donde esté incluído. Por este motivo no se darán más detalles al respecto.

4.3 Especificaciones Z y la relación *uses*

Como dijimos anteriormente, el testing de integración está fuertemente relacionado con el diseño de software. En este trabajo utilizamos la teoría de diseño de D. Parnas introducida en la Sección 2.1. En particular utilizamos el concepto de relación *uses*. La relación *uses* es de relevancia para los métodos de MBT ya que está basada en las especificaciones de las subrutinas.

De aquí en adelante, A y B son las especificaciones Z de las subrutinas P y Q, y además P *uses* Q, a menos que se diga lo contrario. En efecto, P *uses* Q significa que la *especificación* de P dice que se necesita una correcta versión de Q. Desde una perspectiva funcional P y Q podrían ser implementadas en una sola unidad cuya especificación es, aproximadamente, la conjunción de las especificaciones P y Q. Como sea, desde la perspectiva estructural (el diseño) puede ser mejor partir esa unidad en dos de manera que una use a la otra. Se ha mostrado en un ejemplo de esta situación en el Capítulo 3 ($ExtraerOk_1$ en un lado, y $ExtraerOk$ y $CAExtraer$ por otro). Entonces, es importante estudiar cómo escribir las especificaciones de las unidades de implementación para que sea fácil encontrar la relación *uses*. En particular, en el contexto del TTF es necesario estudiarlo para especificaciones Z.

A continuación, formalizamos algunos conceptos para introducir guías en el testing de integración.

Definición 1 *Si S es un sistema, definimos $\overline{S}$ como el conjunto de todas las subrutinas que componen S.*

Definición 2 *Para cualquier sistema S, uses es una relación binaria sobre $\overline{S} \times \overline{S}$ tal que para todo $\mathsf{P}, \mathsf{Q} \in \overline{S}$, P uses Q si y solo si existen situaciones en las cuales el correcto funcionamiento de P depende de la disponibilidad de una correcta implementación de Q [33].*

Definición 3 *Para todo sistema S, calls es una relación binaria sobre $\overline{S} \times \overline{S}$ tal que para todo $\mathsf{P}, \mathsf{Q} \in \overline{S}$, P calls Q si y solo si existe una llamada desde P a Q directa o indirectamente a través de una cadena de llamadas.*

Definición 4 *Para cualquier sistema S y para todo $\mathsf{P}, \mathsf{Q} \in \overline{S}$, se define:*

$$\mathsf{P}\ \overline{uses}\ \mathsf{Q} \mathrel{\widehat{=}} \mathsf{P}\ uses\ \mathsf{Q} \land \mathsf{P}\ calls\ \mathsf{Q}$$

Definición 5 *Si* P $\overline{uses}$ Q *y* Z *es usado como el lenguaje de especificación para* S*, y* A *y* B *son las especificaciones para* P *y* Q*, entonces* A *deberá ser escrita de la siguiente manera:*

$$A \mathrel{\widehat=} \mathcal{SE}(B, A_1, \ldots, A_n) \qquad (\dagger)$$

quedando

$$\mathsf{P}\ \overline{uses}\ \mathsf{Q} \Leftrightarrow A \mathrel{\widehat=} \mathcal{SE}(B, A_1, \ldots, A_n)$$

$\mathcal{SE}$ es una expresión de esquemas que depende de los esquemas B y $A_1, \ldots, A_n$ y B es la especificación de Q. Eso es, la especificación de Q es parte de P la cual es complementada por los esquemas A_i. Un ejemplo de esto es *ExtraerOk* en el Capítulo 3. En el ejemplo, *ExtraerOk* es A, *CajaAhorrosABanco* es A_1, *CAExtraerOk* es B, $\mathsf{Banco} :: \mathsf{extraer}$ es P y $\mathsf{Caja} :: \mathsf{extraer}$ es Q. Si P usa otra subrutina además de Q, entonces sus correspondientes esquemas Z participarán en (†) como B. De aquí en adelante se usará (†) pero los resultados pueden ser extendidos a casos más complejos donde P use más de una subrutina.

Proponemos la siguientes guías para escribir especificaciones Z que serán usadas durante el testing de integración.

- Cada subrutina está especificada por un esquema. Mas precisamente, para toda $\mathsf{P} \in \overline{S}$ hay un nombre de esquema A el cual es su especificación (Sección 4.1).
- Los usuarios deberán generar casos de prueba solo para aquellos esquemas que son especificación de subrutinas que pertenecen a $\overline{S}$ (Sección 4.1).
- Los esquemas de operación deben ser generados siguiendo las consideraciones de reducir los espacios de estados (Sección 4.2).
- Si P *calls* Q pero P *u$\not$ses* Q, entonces B no debe formar parte de A ya que P *u$\not$ses* Q significa que la especificación de P dice que no depende de Q. Entonces incluir B en A sería un error porque esto indicaría una dependencia funcional de P a Q. Un ejemplo de este segundo escenario es cuando $\mathsf{Caja} :: \mathsf{extraer}$ llama a err, discutido en el Capítulo 3.
- Si P *uses* Q pero P *c$\not$alls* Q, entonces B no debe formar parte de A, al menos en lo que respecta al tesging de integración. Éste caso es discutido más adelante en *Errores Globales* en el Capítulo 7.

Capturar las diferencias entre las relaciones *uses* y *calls* en la especificación tiene importantes consecuencias en el testing de integración. Si P *calls* Q, entonces, será necesario un *stub* para Q cuando P sea testeado en soledad. En general, este *stub* debería verificar B (la especificación Q) porque de otra manera P podría parecer erróneo cuando en realidad los errores podrían venir del *stub* Q. Si se asume que P *u$\not$ses* Q, entonces, el *stub* de Q puede ser cualquier cosa que compila con la signatura de Q (aún el mismo Q) porque la correctitud de P no depende de Q (dado que P *u$\not$ses* Q). Por lo tanto, si P *calls* Q pero P *u$\not$ses* Q podemos concluir que cuando es testeado P: (a) el *stub* para Q puede ser generado automáticamente o directamente se puede usar a Q si está disponible; y (b) si el testing de integración revela errores en P los mismos no pueden ser debido a una presencia de una Q incorrecta.

Capítulo 5

Testing de integración en el TTF

En este Capítulo, en la Sección 5.1, mostramos los beneficios de guiar el testing de integración con la relación *uses*. En la Sección 5.2 cómo adaptar la generación de casos de prueba con el TTF para el testing de integración guiado por la relación de *uses*.

5.1 Testing de integración guiado por la relación *uses*

Si P *uses* Q y un error es expuesto durante el testing de P, la misma naturaleza del testing impide restringir la búsqueda de la causa del error a P, ya que depende de al menos Q, que en el mejor de los casos, ya fue testeada, pero no se probó su correctitud. Esta es una de las dificultades más grandes durante el testing de integración puesto que el testeo de subrutinas que usan docenas de otras tienden a exacerbar este problema. Si el testing de integración es guiado por la relación *uses* este problema se minimiza, como se muestra a continuación.

Parnas restringe la relación *uses* a una jerarquía porque de otra manera "se puede terminar con un sistema en el cual nada funciona hasta que todo funciona" [33]. Si se da un ciclo en la secuencia de subrutinas $f_1 \dots f_n$ tal que f_i *uses* f_{i+1}, entonces para que f_i funcione correctamente será necesario que lo hagan las siguientes en la secuencia, y como es un ciclo, se llegará el punto de partida. Es decir, para que cada una de ellas funcione correctamente se necesitan que todas lo hagan; basta que una no funcione para que ninguna lo haga.

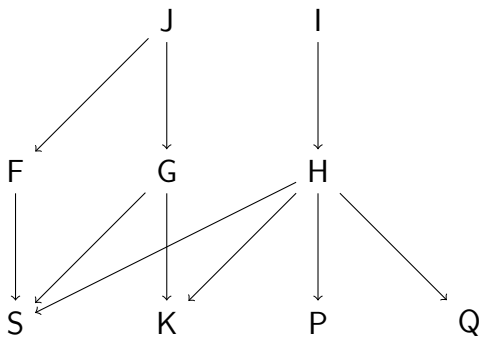

Figura 5.1: Una estructura jerárquica de la relación $\overline{uses}$.

Si *uses* define una jerarquía, existe un conjunto de subrutinas, $\mathfrak{U}_0$, las cuales no usan ninguna otra subrutina. Estas deberían ser las primeras en testearse porque la causa de algún error solo podría estar localizada en ellas mismas. Luego hay otro conjunto de subrutinas $\mathfrak{U}_1$, cuyos elementos sólo $\overline{uses}$ subrutinas de $\mathfrak{U}_0$. Estas deberían ser las segundas en ser testeadas, justo después que las de $\mathfrak{U}_0$ hayan pasado todas sus pruebas. Se sigue con un conjunto $\mathfrak{U}_2$ que sólo $\overline{uses}$ elementods de $\mathfrak{U}_0$ y $\mathfrak{U}_1$ y así.

Ilustramos los beneficios de guiar el testing de integración con la jerarquía que define la relación *uses*, consideramos un sistema compuesto por subrutinas que tienen la siguiente relación: $\mathsf{J}\,\overline{uses}\,\mathsf{G}$, $\mathsf{J}\,\overline{uses}\,\mathsf{F}$, $\mathsf{I}\,\overline{uses}\,\mathsf{H}$, etc. (ver Figura 5.1). Se establecen entonces 3 niveles: $\mathfrak{U}_0 = \{\mathsf{S}, \mathsf{K}, \mathsf{P}, \mathsf{Q}\}$, $\mathfrak{U}_1 = \{\mathsf{F}, \mathsf{G}, \mathsf{H}\}$ y $\mathfrak{U}_2 = \{\mathsf{I}, \mathsf{J}\}$. Primero, supongamos que S y K han pasado todos sus tests pero hay un error, e, en S que no fue descubierto mientras fue testeada. Supongamos además que e inducirá fallas cuando F y G sean testeadas. Tercero, asumamos que los errores son reparados después de hacer todos los tests.

Entonces, el mejor orden para el testing es aquel que reduce el costo de encontrar e, porque todos los otros factores son iguales. Si e es encontrado en el testing de F, e puede estar localizado en F o S. Dado que en el testing de F, esta puede invocar a S de manera que ejecute una traza que no haya sido ejecutada durante el testing de S, justo donde está e. Y como además F requiere el funcionamiento correcto de S, la falla puede manifestarse en el testing de F cuando falle S. Pero si es encontrado al testear G, puede estar localizado en S, K y G, (ver Figura 5.1).

Por otro lado, si seguimos el diseño basado en ocultación de información de Parnas [34] analizado en la Sección 2.1, al limitar el tamaño y complejidad de los módulos, se limita también el tamaño para las subrutinas que lo componen. Es decir, suponemos que el código se tendrá que distribuir en subrutinas con un tamaño acotado, lo que achica las posibilidades de tener un sistema muy heterogéneo en cuanto a la complejidad y tamaño de las subrutinas. Por esto último es mejor testear F antes que G, así será más simple encontrar a e. Dado que, si bien empatan en el hecho de que en ambos casos hay que buscar en S y en sus propios códigos fuentes, en el segundo caso probablemente también haya que buscar en K. Un análisis similar se puede hacer para determinar el orden entre J e I. I usa una subrutina menos que J, pero si se tiene en cuenta la clausura transitiva de *uses*, y e está en S, encontrar e en el primer caso supondría buscar en cinco subrutinas mientras que en el segundo son cuatro.

Otra consideración es que en general las subrutinas de nivel inferior son consideradas más frecuentemente para las búsquedas y reparaciones de errores, por ser susceptibles de ser invocadas más veces y por una mayor cantidad de subrutinas, que las de nivel superior. La probabilidad de que e esté en niveles inferiores debería ser menor. Esto implica (como sucede en la práctica) que una vez producido el error, la búsqueda del mismo no sea azarosa[1]. En el ejemplo, al testear J e I, el nivel inferior (S, K, P y Q) ya pasó dos niveles de testing (con sus posibles reparaciones), cuando se testeó su nivel y cuando se hizo lo propio con el nivel intermedio. Entonces por más que I tenga más subrutinas en total, encontrar el error e podría ser más fácil al testear I ya que solo usa una subrutina de nivel intermedio mientras J usa dos. Es decir, el orden de selección de las subrutinas que favorece la localización de errores, depende de la cantidad de sus descendientes pero también de su distribución en los distintos niveles. Un caso ejemplar es el de las funciones de las bibliotecas estándar de los lenguajes de programación: están en la base de la jerarquía y se les tiene mucha confianza porque las usan millones de usuarios. Entonces si la subrutina W usa 10 funciones de una biblioteca estándar y una de nivel intermedio en tanto que la subrutina V usa una función de biblioteca y 5 de nivel intemedio, convendría testear primero W y luego V a pesar de que W usa más subrutinas que V porque puede, razonable mente, considerarse que las funciones de biblioteca no tienen errores.

Las siguientes definiciones formalizan las ideas dadas en esta sección.

Definición 6 $\mathfrak{U}_0 = \operatorname{ran} \overline{uses} \setminus \operatorname{dom} \overline{uses}$

[1] Caso contrario solo dependería de la cantidad de subrutinas donde buscar.

Definición 7 *Para cada $i > 0$, se define*

$$\mathfrak{U}_i = \{\mathsf{P} \in \overline{S} \bullet (\exists\, \mathsf{Q} \in \overline{S}, j \in 0\,..\,i-1 \bullet \mathsf{Q} \in \mathfrak{U}_j \wedge \mathsf{P}\ \overline{uses}\ \mathsf{Q})\}$$

Ejemplo, si las relaciones fueran las siguientes, $\mathsf{G}\ \overline{uses}\ \mathsf{S}$, $\mathsf{F}\ \overline{uses}\ \mathsf{S}$ y $\mathsf{F}\ \overline{uses}\ \mathsf{G}$, entonces $\mathfrak{U}_0 = \{\mathsf{S}\}$, $\mathfrak{U}_1 = \{\mathsf{G}\}$ y $\mathfrak{U}_2 = \{\mathsf{F}\}$.

Definición 8 *Sea el grafo dirigido $G = (V, E)$ donde $V \mathrel{\widehat{=}} \overline{S}$ y $E \mathrel{\widehat{=}} \overline{uses}$. Se define $\mathfrak{D}(\mathsf{P})$ (descendientes de P) como todos los vértices (subrutinas) $\mathsf{Q} \in V$ tal que existe un camino dirigido de P a Q.*

Definición 9 *Definimos la posición en el proceso de testing de una subrutina P, por la cantidad de subrutinas descendientes $\mathfrak{D}(\mathsf{P})$ y por un valor de confianza $f(i)$ que depende del nivel en el árbol de cada subrutina descendiente, donde N es la altura del árbol.*

$$pos(\mathsf{P}) = \sum_{i=0}^{N} \#(\mathfrak{D}(\mathsf{P}) \cap \mathfrak{U}_i) * f(i) \qquad (\ddagger)$$

El valor de $f(i)$ puede ser determinado empíricamente. Si para todo $i, f(i) = c$ donde c es cualquier valor numérico constante, entonces el orden (ver la definición siguiente) para el testing de todas las subrutinas queda determinado solamente por la cantidad de subrutinas descendientes. Es decir, no se está valorando el hecho de que a los niveles inferiores se les pudieran tener mayor grado de confianza[2]. Para valorar ese hecho, $f(i)$ debería ser una función creciente.

Definición 10 *Entonces se puede establecer un orden para el testing de integración guiado por la relación uses que tienda a minimizar el costo de la búsqueda de los errores de la siguiente manera. La subrutina Q se va a testear antes que P si y solo si: $pos(\mathsf{Q}) < pos(\mathsf{P})$. En caso de ingualdad es indistinto que subrutina se elija.*

5.2 Generación de casos de prueba durante el Testing de Integración

Si van a ser testeadas P y Q usando un método de MBT entonces sus especificaciones, A y B, deben ser analizadas de modo de generar sus casos

[2]Que como dijimos, por la naturaleza misma del proceso de testing *bottom-up* son considerados más frecuentemente en la búsqueda y reparación de los errores que los niveles superiores.

de prueba abstractos. La pregunta es, *¿cómo la relación* $\mathsf{P}\ \overline{uses}\ \mathsf{Q}$*, y por lo tanto el hecho de que A incluya a B, cambiaría la manera estándar en la cual el método MBT es aplicado?* Si el método de MBT analiza en detalle las fórmulas A y B entonces se requerirá alguna adaptación porque de otra manera se expandirá B dentro de A con el significado de que los casos de prueba generados para P estarán influenciados también por Q (ver Figura 5.2).

Como sea, Q ya ha pasado todas sus pruebas como unidad y en principio, no hay ninguna razón para analizarla de nuevo. Más aún, si la clausura transitiva de $\overline{uses}$ incluye una larga cadena de subrutinas empezando desde P, entonces la expansión total de A terminará en una inmensa fórmula la cual será difícil de analizar por cualquier implementación del método MBT.

$$
\begin{array}{|l}
\hline
ExtraerOk\_VIS \\
cajas : NUMCTA \pfun DINERO \times HIS \\
clientes : DNI \pfun NOMBRE \times DOMICILIO \\
titulares : DNI \pfun NUMCTA \\
caja : CajaAhorros \\
n? : NUMCTA \\
d? : DINERO \\
m? : DINERO \\
\hline
d? \in \mathrm{dom}\ clientes \\
n? \in \mathrm{dom}\ cajas \\
titulares\ d? = n? \\
cajas\ n? = caja \\
m? > 0 \\
m? \leq caja.1 \\
\hline
\end{array}
$$

$$
\begin{array}{l}
CAExtraerOk\_VIS == \\
\quad [caja : CajaAhorros;\ m? : DINERO \mid m? > 0 \land m? \leq caja.1]
\end{array}
$$

Figura 5.2: Si *ExtraerOk* (del Capítulo 3) es A y *CajaExtraerOk* es B, entonces el *VIS* de B, *CAExtraerOk_VIS*, queda expandido en el de A, *ExtraerOk_VIS*.

Esto está en consonancia con la idea que durante el testing de integración las unidades ya testeadas deberían tratarse como cajas negras (ver Figura

5.3). Por otro lado, una desventaja de no expandir B en A podría darse en el caso de que Q no sea testeado tan exhaustivamente como si B fuese expandido. Es decir, al expandir B en A los casos generados a partir de A podrían hacer que P invoque a Q de formas diferentes a cuando Q fue testeado y por lo tanto se descubrirían más errores. Este punto es discutido en la Sección 7, analizado con un ejemplo en la Sección 8 y explicado su uso con Fastest en los dos apéndices finales.

$$
\begin{array}{|l}
\hline
\mathit{ExtraerOk\_VIS} \\
\mathit{cajas} : \mathit{NUMCTA} \rightarrow \mathit{DINERO} \times \mathit{HIS} \\
\mathit{clientes} : \mathit{DNI} \pfun \mathit{NOMBRE} \times \mathit{DOMICILIO} \\
\mathit{titulares} : \mathit{DNI} \pfun \mathit{NUMCTA} \\
\mathit{caja} : \mathit{CajaAhorros} \\
n? : \mathit{NUMCTA} \\
d? : \mathit{DNI} \\
\mathit{CAExtraerOk\_Decls} \\
\hline
d? \in \mathrm{dom}\ \mathit{clientes} \\
n? \in \mathrm{dom}\ \mathit{cajas} \\
\mathit{titulares}\ d? = n? \\
\mathit{cajas}\ n? = \mathit{caja} \\
\mathrm{pre}\ \mathit{CAExtraerOk} \\
\hline
\end{array}
$$

Figura 5.3: Si *ExtraerOK* (del Capítulo 3) es A y *CajaExtraerOk* es B, entonces el *VIS* de B, *CAExtraerOk_VIS*, queda contraído en el de A, *ExtraerOk_VIS*.

Nuestra propuesta para generar casos de prueba de integración mediante el TTF consiste en lo siguiente. Primero el TTF se aplica a elementos que pertenecen a $\mathfrak{U}_0$ sin ningún cambio [11]. Si $\mathsf{P} \in \mathfrak{U}_1$, entonces por lo que se explicó en 4.1, la especificación A debería ser compuesta: $A \mathrel{\widehat{=}} \mathcal{SE}(B, A_1, \ldots, A_n)$ para algún B tal que éste es la especificación de algún $\mathsf{Q} \in \mathfrak{U}_0$. En este caso, cuando el TTF es aplicado a A, B no se expande, contradiciendo la presentación original tanto del TTF como de Fastest. Solo las variables declaradas en B y referenciadas por algún A_i son exportadas desde B a A, por razones de consistencia. Esto implica que los casos de prueba para P son generados solamente analizando A. Es decir, solo se analiza la estructura de $\mathcal{SE}$ y los predicados en $A_1, \ldots, A_n$. En otras palabras, B influye la generación de casos para A sólo como un todo y por su lugar en $\mathcal{SE}$. Esto significa que el TTF generará, como mínimo, casos que van a

hacer que P llame a Q desde diferentes lugares y con diferentes parámetros.

Por ejemplo, analicemos la especificación del Capítulo 3, Figura 3.2. La idea es que ya se ha testeado Caja :: extraer y queremos testear Banco :: extraer, es decir que ya se han generado casos para *CAExtraerOk* y ahora se hará lo mismo con *ExtraerOk*. Luego si aplicamos la táctica DNF [11] al esquema *ExtraerOk*, dejando contraído *CAExtraerOk* como estamos proponiendo (ver Figura 5.3), habrá casos que harán que se pruebe Banco :: extraer con un *DNI* que pertenezca a los clientes, un número de cuenta válido para ese *DNI*, etc. O sea, estos casos probarán si Banco :: extraer implementa correctamente $d? \in \mathrm{dom}\ clientes \land n? \in \mathrm{dom}\ cajas \land titulares\ d? = n? \land cajas\ n? = caja$ y si llama a Caja : extraer cuando debería. En algún sentido, esto es todo lo que vale la pena testear de Banco : extraer dado que la correctitud del algoritmo de extracción del método Caja :: extraer ya fue probada. La variable $m?$ que representa el monto a extraer queda libre en *ExtraerOk* pues, a los efectos de probar Banco :: extraer, no importa cómo queda el saldo en la caja, sólo importa cómo queda la caja en relación al cliente, a los titulares y a las otras cajas. Además, si también se aplica la táctica SP [11] a Banco : extraer, por ejemplo a la expresión $d? \in \mathrm{dom}\ clientes$ se obtendrían clases de prueba como las siguientes:

$$ExtraerOk_1^{DNF} \mathrel{\widehat{=}} [ExtraerOk^{VIS} \mid d? \in \mathrm{dom}\ clientes \land n? \in \mathrm{dom}\ cajas \land titulares\ d? = n? \land cajas\ n? = caja]$$

$$ExtraerOk_1^{SP} \mathrel{\widehat{=}} [ExtraerOk_1^{DNF} \mid \mathrm{dom}\ clientes = \{d?\}]$$

$$ExtraerOk_2^{SP} \mathrel{\widehat{=}} [ExtraerOk_2^{DNF} \mid \mathrm{dom}\ clientes \neq \{d?\} \land d? \in \mathrm{dom}\ clientes]$$

Como puede verse estas condiciones testean la correctitud de la implementación de $d? \in \mathrm{dom}\ clientes$, que es parte de la responsabilidad de Banco : extraer y no de Caja : extraer, por lo que no hubiera aportado nada expandir *CAExtraerOk* en *ExtraerOk*.

Capítulo 6

Subrutinas como *stubs* de ellas mismas

La distinción entre las relaciones *uses* y *calls* reduce la necesidad de crear manualmente *stubs* (último parrafo en la Sección 4.3). También hace a esta reducción la naturaleza bottom-up del testing guiado por $\overline{uses}$ (Sección 5.1). Como sea, un *stub* para Q es necesario cuando P $\overline{uses}$ Q y se va a testear P. Una manera de evitar la creación del *stub* para Q podría ser usar la misma Q, pero esto, en general, no puede hacerse ya que no se ha demostrado que Q sea correcta, solamente fue testeada. Sin embargo, si Q ha pasado algunos tests entonces podemos estar seguros que se comporta correctamente para esas entradas. Ahora, si P es testeada de manera que siempre llame a Q tal cual como fue testeada, entonces se puede usar Q como *stub*. Además, la causa de un error encontrado durante el testing de P puede ser solamente culpa del mismo P ya que ha sido "probada la correctitud" de Q para esas entradas.

Hemos hecho un intento de formalización de estas ideas, creando las bases para la mecanización de la búsqueda de subrutinas que pueden ser *stubs* de ellas mismas. Definimos y probamos dos teoremas en [15] que dan las condiciones para que las subrutinas sean usadas como *stubs*. Ambos teoremas asumen (†) y sus pruebas se basan en la hipótesis de uniformidad, la cual puede ser enunciada de la siguiente manera [26, página 17].

Sea S una especificación cuya implementación es P, y C un subconjunto del espacio de entrada de S y $t \in C$. Sea $CORRECT(\mathsf{P}, S, t)$ un predicado que se interpreta como: P se comporta correctamente para t con respecto a S. También definimos:

$$CORRECT(\mathsf{P}, S, C) \equiv (\forall\, x \in C \bullet CORRECT(\mathsf{P}, S, x))$$

Entonces, la hipótesis de uniformidad vale si y solo si:

$$\forall\, x \in C \bullet (CORRECT(\mathsf{P}, S, x) \Rightarrow CORRECT(\mathsf{P}, S, C))$$

Con el fin de enunciar los teoremas se necesita definir algo de notación. Consideremos los esquemas $A, A_1 \ldots A_n$ y B como en (†). De acuerdo a la Sección 5.2 solo $A_1, \ldots, A_n$ serán expandidos en A.

Sea el esquema $AI = A_1 \wedge A_2 \wedge \ldots \wedge A_n$ y sea $vars(AI)$ el conjunto de variables declaradas en el esquema AI^{VIS} y $vars(B)$ el conjunto de variables declaradas en el esquema B^{VIS}. Si a es un caso de prueba derivado del esquema A, entonces $B^A(a)$ significa la sustitición en B de las variables $vars(B) \cap vars(AI)$ por los mismos valores de las variables en a.

Notar que si a fue generado siguiendo las condiciones de 5.2, es decir, con B contraído en A, las variables que pertenecen a $vars(B)$ tendrán en $B^A(a)$ valores que fueron generados sin tener en consideración las precondiciones de B.

Ejemplo 6.1 *Sea* $A = A_1 \wedge B$, $A_1 = [a_1, x, x' : \mathbb{Z} | x > 0]$, $B = [b_1, x, x' : \mathbb{Z} | b_1 > 0]$ *y* $a = [a_1, x, b_1 : \mathbb{Z} | a_1 = -1 \wedge x = 1 \wedge b_1 = -1]$ *entonces* $B^A(a) = [b_1, x, x' : \mathbb{Z} | x = 1 \wedge b_1 > 0]$.

Se notará $A^A(a)$ como $A(a)$. Si A y B participan en (†), entonces $vars(B) \cap vars(AI) \neq \emptyset$, porque si este no fuera el caso, B no podría influenciar a A. Aunque tampoco siempre se da el caso donde $vars(B) = vars(AI)$ (ver Figura 6.1).

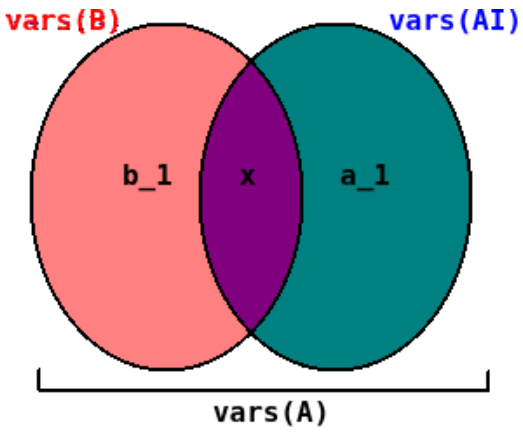

Figura 6.1: Relación entre $vars(A)$, $vars(B)$ y $vars(AI)$ para el ejemplo anterior.

El Teorema 1 asume que a nivel de la especificación, no hay cambio de estado en P antes de llamar a Q. Este es el caso de *ExtraerOk* en la Sección 3.

Teorema 1 *Sean* P *y* Q *dos subrutinas tales que* P $\overline{uses}$ Q *y sean* A *y* B *sus especificaciones Z, las cuales a su vez cumplen con* (†). *Asumir que hay solo un cambio de estado en* A. *Sean* $B_1, \ldots, B_n$ *las hojas del árbol generado al aplicar el TTF a* B. *Sea* a *un caso de prueba para* P *derivada de* A. *Si existe un* B_j *tal que* $B_j^A(a) \neq \emptyset$ *y además* Q *ha pasado los test derivados de* B_j, *entonces* Q *puede ser usado como stub cuando* P *es testeado en* a.

Demostración *Si hay un único cambio de estado en* A *entonces* Q *se ejecuta con los mismos valores que* P *para variables en* $vars(B) \cap vars(AI)$. *Si* $B_j^A(a) \neq \emptyset$, *entonces existe* $b \in B_j$ *tal que* a *y* b *son iguales en las variables* $vars(AI) \cap vars(B)$. *Si* Q *ha pasado el test que determina* B_j, *por la hipótesis de uniformidad,* Q *es también correcto en* $b \in B_j$. *Por lo tanto, cuando* P *es ejecutado en* a, Q *es ejecutado en* b, *y así retorna una respuesta correcta a* P. *Entonces* Q *puede ser usada como stub cuando* P *es testado en* a.

En este ejemplo mostraremos la aplicación del Teorema 1.

Ejemplo 6.2 *Sea* $A == A_1 \wedge B$ *tal que:*

$$A_1 == [a_1, x : T | P_{A_1}(a_1, x)]$$
$$B == [b_1, x : T | P_B(b_1, x)]$$

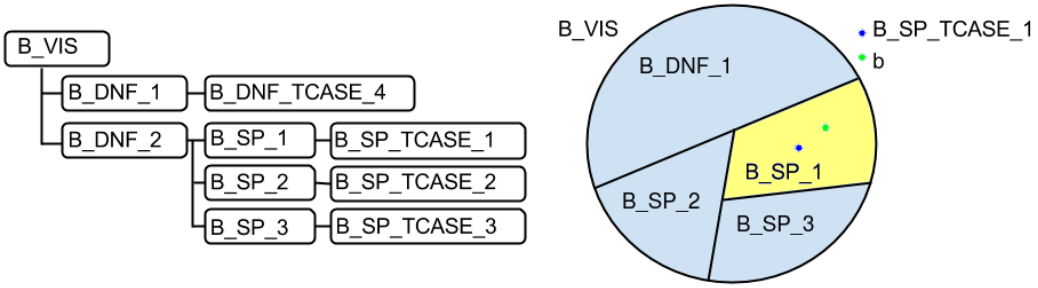

Figura 6.2: Árbol de prueba y partición del conjunto *VIS* del esquema B para el ejemplo del Teorema 1.

Si Q *ha pasado la prueba para el caso* $B\_SP\_TCASE\_1$ *(ver Figura 6.2), se asume entonces que* Q *funciona bien para todos los casos de la clase* B_SP_1. *Si además tenemos, por ejemplo, un caso* $a == [a_1, b_1, x : \mathbb{Z} | a_1 = c_1, b_1 = c_2, x = c_3]$, *y* $B\_SP\_1^A(a) \neq \emptyset$ *entonces existe* b *tal que* $b \in B\_SP\_1$ *y* $x = c_3$ *en* b. *El teorema dice que* Q *se va a comportar bien en* b, *ya que* b *y* $B\_SP\_TCASE\_1$ *pertenecen a la misma clase, en cuyos casos* Q *ejecuta bien. Finalmente se puede usar a para testear* P, *porque cuando se ejecute* P *en* a, Q *se ejecutará en* b *donde sabemos que funciona bien. O sea* Q *puede ser su propio stub cuando* P *se ejecuta en* a.

El siguiente teorema aplica cuando el esquema A especifica dos cambios de estados, uno de los cuales está en B. Es decir que si B especifica Q, como fue recomendado en la Sección 4.1, entonces habrá un cambio en P y otro en Q. Este es el caso cuando se utiliza el operador de composición de esquemas $\overset{\circ}{9}$. Por eso, si Q depende del nuevo estado de las variables, se usarán sus nuevos valores como entrada (los valores de las variables de estado que deja P). Por lo tanto, se necesita saber si estos nuevos valores fueron usados para testear Q. Esta es la razón para pasar de $B_j^A(a)$ en el Teorema 1 a $B_j^A(a')$ en el Teorema 2, donde $a' \in A(a)$.

Teorema 2 *Sean* P *y* Q *dos subrutinas tal que* P $\overline{uses}$ Q *y sean* A *y* B *sus especificaciones Z, las cuales a su vez cumplen con* (†). *Se asume que* B *está siempre del lado derecho del operador composición* ($\overset{\circ}{9}$). *Sean* $B_1, \ldots, B_n$ *las hojas del árbol de prueba generado al aplicar el TTF a* B. *Sea* a *un caso de prueba para* P *y sea* a' *cualquier elemento en* $A(a)$. *Si existe un* B_j *tal que* $B_j^A(a') \neq \emptyset$, *entonces* Q *pueden ser usados como stub cuando* P *es testeado*

en a. Se asume además que Q *ha ejecutado exitosamente los test derivados de* B_j.

Demostración *Si a es un caso de prueba para* P *entonces solo hay variables de estado primadas en $A(a)$. Si B está siempre del lado derecho del operador* $\fatsemi$ *entonces habrá una transición de estado antes de ejecutar B. Entonces, si $a' \in A(a)$, B usará los valores en a' para realizar la transición. Si $B_j^A(a') \neq \emptyset$, entonces existe $b \in B_j$ tal que a' y b son iguales en las variables $vars(AI) \cap vars(B)$, así la demostración fácilmente se sigue de la demostración del Teorema 1.*

Estos teoremas son discutidos en las Sección 9.1. Para desarrollar algunos resultados prácticos en la tesina, automatizamos la aplicación del Teorema 1 en el programa Fastest, la analizamos en detalle en el Apéndice B y la aplicamos a un caso de estudio en la Sección 8.

Capítulo 7

Detección de Errores durante el Testing de Integración

Leung y White dan una clasificación de errores que pueden ser detectados durante el testing de integración [30, 31]. En esos trabajos los autores tratan de hacer una distinción entre los errores que podrían ser detectados durante el testing de unidad y aquellos que son específicos del testing de integración. Más abajo brevemente explicamos cómo el TTF extendido al testing de integración puede detectar cada uno de estos errores. Cada caso esta acompañado con un simple ejemplo artificial para graficar lo definido. En todos los ejemplo, a menos que se diga lo contrario, P es la subrutina especificada por el esquema A y Q la subrutina especificada por B.

Errores de interpretación Están divididos en tres subclases.

- Errores de mal funcionamiento (Wrong function errors, WFE). Q no provee la funcionalidad indicada por su especificación y P no lo sabe.

  Dado que el TTF (y otros métodos de MBT) genera casos de prueba para Q desde su especificación, los WFE serán detectados cuando Q sea testeado como unidad. En otras palabras, si un caso de prueba de Q generado por el TTF encuentra un error en Q, lo que indica es que no provee la funcionalidad indicada por su especificación.

  Ejemplo 7.1 *Es el caso más común del TTF. Tenemos la siguiente especificación:*

  $$A == [e?, x, x' : \mathbb{Z} | x' = e?/x]$$

  que fue implementada, erróneamente, de la siguiente manera:

$$\textbf{\textit{void}}\ \ p(\textbf{\textit{int}}\ \ e)\{x\ =\ x*e;\}$$

Con el TTF[1] *se puede generar el siguiente caso:*

$$A_{30}^{TC} \mathrel{\widehat{=}} [A_{30}^{NR} \mid e? = 3 \land x = 3]$$

que refina a[2]*:*

$$\{e\ =\ 3,\ \ x\ =\ 3\}$$

El programa P *deja las variables de la siguiente manera:*

$$\{e\ =\ 3,\ \ x\ =\ 9\}$$

como x es una variable de estado, la abstracción queda $x' = 9$, mientras que la evaluación del caso abstracto en la operación A deja a $x' = 1$. Entonces la comprobación (Sección 2.2) dara falso, lo quiere decir que se descubre el error.

- Errores de funcionalidad extra (Extra function errors, EFE). Q provee más funcionalidad que la que P necesita. Los desarrolladores de P saben esto pero implementan mal P haciendo que llame la funcionalidad extra. El TTF generará como mínimo un caso para cada una de las funcionalidades en la especificación de Q. Por ejemplo, van a ser de mucha utilidad las tácticas de pruebas como DNF y FT [11]. Si P es testeado de tal manera que Q sea ejecutado en todas esas funcionalidades, entonces los problemas de P aparecerán. En otras palabras, es necesario aplicar el TTF a A de tal manera que genere suficientes casos de prueba para que P llame a Q y este último ejecute todas sus funcionalidades.

  Si es el caso de (†) tal que B especifica Q como se recomienda en 4.1, entonces B define esa funcionalidad extra y Q tiene que implementarla. Entonces es en algún A_i donde está necesariamente la restricción a no usar todos los servicios de Q. Esto significa que la precondición de A_i debe aceptar el caso que ejecute la funcionalidad, pero debe tener una postcondición que especifique una salida o variación de estado diferente a que si la funcionalidad fuese usada, usualmente podría ser un esquema

[1] Usando Fastest y aplicando la táctica *Numeric Ranges* (NR) [9].

[2] Estrictamente deberían ser asignaciones de un lenguaje destino, en este caso es uno irreal estilo C, pero a los efectos del ejemplo alcanza.

de error. En otras palabras, se necesita contar con una especificación total de P[3], y generar los casos desde ahí.

A continuación introducimos un teorema definido en [15] que explica cómo se pueden detectar los EFE a través del TTF.

Teorema 3 *Sean* P *y* Q *dos subrutinas tales que* $\mathsf{P}\,\overline{uses}\,\mathsf{Q}$ *y sean* A *y* B *sus especificaciones* Z*, las cuales a su vez cumplen con* (†)*. Se asume que hay solo un cambio de estado en* A*. Sean* $B_1, \ldots, B_n$ *las hojas del árbol generado al aplicar el TTF a* B*. Se asume además que las hojas representan toda la funcionalidad propuesta por* Q*. Sean* $a_1, \ldots, a_m$ *los casos para* P *derivados de* A*. El TTF detectará todo EFE en* P *si para cada* $j \in 1\,..\,n$ *existe* $i \in 1\,..\,m$ *tal que* $B_j^A(a_i) \neq \emptyset$ *y además* Q *ha pasado los test derivados de* B_j*.*

Demostración 1 *Si las hojas* $B_1, \ldots, B_n$ *representan toda la funcionalidad de* Q*, entonces existe* $B_f \in \{B_1, \ldots, B_n\}$*, que representa la funcionalidad extra que* P *no debe usar.*

Además, como para cada $j \in 1\,..\,n$ *existe* $i \in 1\,..\,m$ *tal que* $B_j^A(a_i) \neq \emptyset$*, en particular existe* a_f *tal que* $B_f^A(a_f) \neq \emptyset$*. Como* Q *ha pasado todos sus test, y por el Teorema 1, existe* $b_f \in B_f$ *tal que* a_f *y* b_f *son iguales en las variables* $vars(A) \cap vars(B)$*, y cuando* P *es ejecutado en* a_f*,* Q *es ejecutado en* b_f*. Significa que* a_f *va a hacer que se ejecute algo en* Q*, desde* P*, que no debería ejecutarse. Esto implica que* P *está usando funcionalidad de* Q *que no debería utilizar. O sea que con* a_f *se va a ver que* P *no cumple con* A*.*

Ejemplo 7.2

$$E ::= E1 | E2$$

$$B1 == [e? : E;\ b, b' : \mathbb{Z} | e? = E1 \land b' = 1]$$

$$B2 == [e? : E;\ b, b' : \mathbb{Z} | e? = E2 \land b' = 10]$$

$$B == B1 \lor B2$$

$$AE == [e? : E;\ y, y' : \mathbb{Z} | e? = E2 \land y' = y]$$

$$AOk == [e? : E;\ b!, y, y', b, b' : \mathbb{Z} | e? = E1 \land y' = b']$$

$$A == (AOk \land B) \lor AE$$

[3]Una especificación total es aquella que define el funcionamiento de la operación para cada uno de los valores posibles del espacio de entrada. Generalmente esto sucede cuando se especifica el esquema de error para la subrutina.

Observar que cuando se da AE no se invoca a Q*. Sin embargo, la implementación es errónea porque* P *llama a* Q *siempre:*

```
int b;
int q(E e){
        if (e==E1) b = 1;
        if (e==E2) b = 10;
        return b;
}
int p(E e){
        y=q(e);
        return y;
}
```

Notar que P *está mal si no se bifurca con la condición sobre el valor de e. Es decir,* Q *ofrece más funcionalidad porque funciona tanto para* $e = E1$ *como para* $e = E2$ *y* P *usa las dos funcionalidades cuando debería usarla solo para* $e = E1$*. Para poder efectivamente testear este error es necesario contar con el esquema AE ya que es desde allí que se podrá ejecutar la funcionalidad extra de* Q*. El error se encuentra al generar un caso desde A con el TTF usando DNF:*

$$A_1^{DNF} \mathrel{\widehat{=}} [A^{VIS} \mid e? = E2 \wedge y = -2147483648]$$

que refina a:

$$\{e = E2,\ y = \_2147483648\}$$

El programa P *deja las variables de la siguiente manera:*

$$\{e = E2,\ b = 10,\ y = 10\}$$

como y es una variable de estado, la abstracción queda $y' = 10$*, mientras que la evaluación del caso abstracto en la operación A deja a* $y' = -2147483648$*. Entonces la comprobación (Sección 2.2) dara falso.*

En [15] se menciona que el teorema similar pero con cambio de estado puede ser demostrado si Q es llamado después de que P haga un cambio de estado. A continuación lo desarrollamos.

Teorema 4 *Sean* P *y* Q *dos subrutinas tales que* $\mathsf{P}\ \overline{uses}\ \mathsf{Q}$ *y sean A y B sus especificaciones Z, las cuales a su vez cumplen con (†). Se asume que B está siempre del lado derecho del operador composición (⨾). Sean* $B_1, \ldots, B_n$ *las hojas del árbol generado al aplicar el TTF a B. Sean*

$a_1, \ldots, a_m$ los casos para P *derivados de A y sean $a'_1, \ldots, a'_m$ donde cada a'_i es cualquier elementos de $A(a_i)$. Se asume que las hojas representan toda la funcionalidad propuesta por* Q*. El TTF detectará todo EFE en* P *si para cada $j \in 1 \ldots n$ existe $i \in 1 \ldots m$ tal que $B_j^A(a'_i) \neq \emptyset$, y además* Q *ha pasado los test derivados de B_j.*

Demostración 2 *Si las hojas $B_1, \ldots, B_n$ representan toda la funcionalidad de* Q*, entonces existe $B_f \in \{B_1, \ldots, B_n\}$, que representa la funcionalidad extra que* P *no debe usar. Además, como para cada $j \in 1 \ldots n$ existe $i \in 1 \ldots m$ tal que $B_j^A(a'_i) \neq \emptyset$, en particular existe a'_f tal que $B_f^A(a'_f) \neq \emptyset$. Como* Q *ha pasado todos sus test, y por el Teorema 2, existe $b_f \in B_f$ tal que a'_f y b_f son iguales en las variables $vars(A) \cap vars(B)$, y cuando* P *es ejecutado en a_f,* Q *es ejecutado en b_f. Asi la demostración se sigue fácilmente de la demostración del Teorema 3.*

- Errores por falta de funcionalidad (Missing function errors, MFE). Las entradas usadas por P llaman a Q fuera del dominio de Q haciendo que se comporte impredeciblemente.

  Si B es total entonces P no puede hacer que Q se comporte impredeciblemente porque existe un comportamiento especificado para cada entrada de Q. Si B es parcial entonces P debería llamar a Q con $b \notin B^{VIS}$ para ejecutarlo fuera del dominio de entrada. Pero se debe encontrar desde b la entrada para P, a, que haga que Q sea llamado con b. Es más fácil calcular a si A realiza solo un cambio de estado. Se define una nueva táctica de prueba, llamada MF, que debería ser aplicada a operaciones cuyas correspondientes subrutinas están en el dominio de la relación *uses*. Las clases de pruebas generadas por MF son: $A_1^{MF} == [A^{VIS} \mid \exists\, x_1, \ldots, x_n \bullet \text{pre } B]$ and $A_2^{MF} == [A^{VIS} \mid \exists\, x_1, \ldots, x_n \bullet \neg \text{ pre } B]$, donde $x_1, \ldots, x_n$ son variables declaradas en B pero no en A. Notar que MF es aplicada a A, no a B pero B es parte de A como en (†)

  Para poder cerrar el cliclo del MBT (ver Figura 2.1) y detectar el MFE, es necesario contar con información extra que determine si efectivamente P llama a Q fuera del dominio, por ejemplo, si se produce un *segmentation fault*, es una cuestión que no se puede abstraer (tal como fue introducido el TTF) entonces por el simple hecho que suceda se sabe que hay un error en P. El ciclo del MBT, para este tipo de

errores, no es tan fácil de modelar y ejemplificar cómo en los casos anteriores, es necesario un análisis más profundo sobre la representación de la ejecución de los programas, que tenga en cuenta además los efectos colaterales de difícil abstracción como el *segmentation fault*. Esto escapa ampliamente a este informe, y sólo diremos que, si bien es posible detectar con el TTF los MFE, en este punto es necesario un estudio más completo y profundo en la etapa de abstracción del MBT para completar el ciclo.

Errores de invocación (Miscoded Call errors). P invoca a Q desde lugares equivocados. Se divide en tres subclases.

- Instrucción de invocación extra (Extra call instruction, ECI). La invocación está en un camino en el cual no debería estar.
- Instrucción de invocación mal localizada (Wrong call instruction placement, WCI). La llamada está en el camino correcto pero en un lugar incorrecto.
- Instrucción de invocación faltante (Missing instruction, MIC). Falta de la instrucción de invocación en un camino dado.

La detección de estos errores es una de las razones para definir A como (†). Si A especifica exactamente todas las llamadas que P debería hacer a Q, entonces el TTF ayudará a descubrir todos estos errores. En efecto, la aplicación de DNF a $\mathcal{SE}(B, A_1, \ldots, A_n)$ generará clases de prueba para cada una de las situaciones donde Q es llamado y donde no; otras tácticas, como tipos libres (FT) (Sección 2.2, pág 11) o cuantificaciónes universales (Universal Quantications, UQ) [18] , completan el trabajo ya que generan condiciones más detalladas donde Q es llamado.

Ejemplo 7.3

$$E ::= E1|E2$$

$$B == [e? : E;\ b, b' : \mathbb{Z} | b' = 17]$$

$$AOk == [e? : E;\ b, b', y, y' : \mathbb{Z} | e? = E1 \land y' = b']$$

$$AE == [e? : E;\ y, y' : \mathbb{Z} | e? = E2 \land y' = y]$$

$$A == (AOk \land B) \lor AE$$

Una mala implementación de P *puede ser la siguiente:*

```
int p(E e){
        if (e == E2){
                y=q(e);
        }
        return y;
}
```

La implementación de P *tiene dos errores, un ECI, porque invoca a* Q *cuando* $e? = E2$ *y la especificación (AE) dice que bajo esa condición no debe hacerlo, y no la invoca si* $e? = E1$ *cuando la especificación dice que tiene que hacerlo (*$AOk \wedge B$*). Se genera el siguiente caso, usando la táctica DNF:*

$$A_2^{DNF} \mathrel{\widehat{=}} [A^{VIS} \mid e? = E2 \wedge y = -2147483648]$$

cuyo refinamiento es el siguiente:

$$\{e = E2,\ y = \_2147483648\}$$

El programa P *deja las variables de la siguiente manera:*

$$\{e = E2,\ y = 17\}$$

como y *es una variable de estado, la abstracción queda* $y' = 17$*, mientras que la evaluación del caso abstracto en la operación* A *deja a* $y' = -2147483648$*. Entonces la comprobación (Sección 2.2) dará falso. Cabe preguntarse qué pasa si el TTF genera el caso de prueba con* $y = 17$*. Es el único caso donde el programa funciona correctamente según la especificación, entonces el proceso no encontrará el error, a pesar de que sabemos que para el resto de los casos si lo hace. Esta desventaja es una característica inherente al proceso del testing, para disminuir las probabilidades de que ocurra, simplemente lo que hace falta es aumentar los casos de prueba aplicando más tácticas. Bastaría con aplicar Numeric Ranges (NR) [9] sobre* y*.*

El error detectado es el ECI, para detectar el MIC basta usar el siguiente caso:

$$A_1^{DNF} \mathrel{\widehat{=}} [A^{VIS} \mid e? = E1 \wedge y = -2147483648]$$

cuyo refinamiento es el siguiente:

$$\{e = E2,\ y = \_2147483648\}$$

El programa P *deja las variables de la siguiente manera:*

$$\{e = E2,\ y = \_2147483648\}$$

la abstracción quedaría $y' = -2147483648$*, mientras que la evaluación del caso abstracto en la operación* A *deja a* $y' = 17$*, la comprobación dará falso.*

Errores de interface (Interface errors, IER). Aparecen cuando hay una violación de interfaz entre dos unidades, como: parámetros con el tipo equivocado, parámetros en distinto orden, o con el formato equivocado, etc. Estos errores son difíciles de detectar por un método de MBT porque las características que los producen son raramente incluidas en una especificación funcional. Más aún, si el lenguaje es fuertemente tipado la mayoría de estos errores son detectados por el compilador [32]. Por estas razones, el TTF extendido al testing de integración puede ayudar a detectar estos errores tanto como cualquier otro método de MBT.

Ejemplo 7.4

$$B == [a?, b?, s! : \mathbb{Z} | s! = a?/b?]$$

$$A == B \land [a?, b?, x' : \mathbb{Z} | b? > a? \land x' = s!]$$

```
void q(int a, int b){
        return a/b;
}
int p(int a, int b){
        if (b>a){
                x=q(b,a);
                return x;
        }
}
```

El error es que se está llamando a Q *con los parámetros en distinto orden a como indica la especificación. Es un error difícil de detectar en tiempo de compilación ya que los parámetros tienen el mismo tipo. Pero puede ser detectado con el TTF usando el siguiente caso:*

$$A_{30}^{NR} \mathrel{\widehat{=}} [A_{30}^{NR} \mid a? = 1 \land b? = 2]$$

cuyo refinamiento es el siguiente:

$$\{a=\ 1\ ,\ b\ =2\}$$

El programa P *deja las variables de la siguiente manera:*

$$\{x=2\}$$

entonces la abstracción de la salida deja a $x' = 2$ *y la evaluación del caso abstracto* $x' = 0$. *Finalmente la comprobación dara falso.*

Errores globales (Global errors, GER) Están relacionados con el mal uso de variables globales [30]. Si P *uses* Q pero P *calls* Q, significa que interactúan a través de un recurso compartido que puede ser pensado como una variable global, g. En este caso Q define un valor para g que luego será usado por P. Si éste valor no es lo que P espera, entonces P puede fallar. Hay dos causas que pueden hacer que P encuentre un valor inesperado en g: (a) Q no verifica B; o (b) Q verifica B pero P asume que Q implementa otra especificación, por ejemplo $\widehat{B}$.

Analizando cómo el TTF puede detectar GER se asume que P *uses* Q pero P *calls* Q, porque cuando P *calls* Q, se aplican todos los resultores anteriores. Si (a) es la causa del error, entonces se reduce a WFE porque es chequear si Q verifica su especificación. Por lo tanto, el verdadero problema del testing de integración referido a variables globales está dado bajo las siguientes condiciones: P *uses* Q pero P *calls* Q y Q verifica B pero P asume que Q implementa una especificación diferente, $\widehat{B}$. Una manera posible para detectar estos errores es ejecutando Q antes de P mientras se testea P. Esta manera, complica el testing de P porque es necesario correr otra unidad antes, y debe ser ejecutada de tal manera que haga fallar a P.

Por eso, proponemos detectar estos errores mediante la *verificación* de la especificación en vez de hacerlo en el testing. En efecto, el problema es un desajuste a nivel de especificación, causando errores a nivel de implementación. Esto es, A asume $\widehat{B}$ en vez de B, entonces el problema es encontrar esa falsa suposición. Si se prueba que la operación involucrada verifica algunas propiedades (invariantes de estado, por ejemplo) entonces esta suposición se detectará. En este sentido, se cambiará B por $\widehat{B}$ convirtiéndose en errónea la especificación de Q. En este caso, $\widehat{B}$ no puede ser incorrecto con respecto a A, porque las propiedades probadas actúan como una base de consistencia común entre ellas. Entonces, si Q verifica $\widehat{B}$ no puede setear un valor incorrecto para g desde la perspectiva de P. Desde aquí todo se reduce a asegurar que P y Q implementen sus especificaciones lo cual significa hacer un testing de unidad exhaustivo para cada uno. Por eso en la Sección 4.3 se propone no incluir B en A cuando P *uses* Q pero P *calls* Q.

Capítulo 8

Caso de estudio

En esta sección aplicamos los resultados a una especificación de un banco [13] levemente adaptada, una versión más completa que la de la Sección 3. La intención es doble: a) mostrar que una buena cobertura es alcanzada para subrutinas en el dominio de la relación *uses* sin mirar los detalles de las subrutinas de las cuales dependen; y b) que los teoremas 1 y 2 pueden ser automáticamente aplicados en varias situaciones, los cuales reducen la necesidad de la creación manual de *stubs*. La realización de un testing exhaustivo[1] de las subrutinas involucradas en este caso de estudio está más allá del alcance de este informe (ejemplos similares han sido discutidos en otros artículos [11]). El proceso de generación de casos de prueba sigue los conceptos presentados en la Sección 2.2 aunque aquí se muestra cómo se realiza en Fastest.

La funcionalidad a implementar son algunas operaciones elementales que le brinda un banco a sus clientes, como son las de alta de un cliente, la baja de una caja para un cliente, la consulta del saldo, el depósito y la extracción de dinero. Solamente hicimos un análisis de la operación de extracción, porque a los efectos de mostrar la integración, es la más completa.

[1] En el apéndice A se muestra el resto del árbol y de los casos de prueba que fueron generados y que no fueron incluídos en esta sección por limpieza en el análisis hecho.

Module	**Banco**
imports	Caja,Cajas,Dinero,Dni,Ncta,Cliente,Mensajes
exportsproc	extraer(**i** NCta, **i** Dinero, **i** Dni) pedirSaldo(**i** Ncta, **i** Dni) depositar(**i** Ncta, **i** Dinero) cerrarCaja(**i** Ncta, **i** Dni) nuevoCliente(**i** Cliente) entrada(**i** NCta, **i** Dni,**i** Dinero):Mensajes
private	validarCliente(**i** NCta, **i** Dni):Mensajes validarMonto(**i** Dinero):Mensajes
comments	En el método extraer(), el cliente identificado por el Dni extrae de la cuenta NCta un monto de Dinero.

Module	**Caja**
imports	Dinero,Historial
exportsproc	saldo():Dinero depositar(**i** Dinero) extraer(**i** Dinero) setMontoMax(**i** Dinero) getHistorial():Historial
comments	En el método extraer() se chequea si el monto ingresado es de posible extracción, si es menor al total de la caja y si es mayor a cero. Además mantiene o actualiza el historial de transacciones.

Module	**Cajas**
imports	Dinero,Caja
exportsproc	delCaja(**i** Caja) actualizarCajas(**i** NCta)
comments	El método actualizarCajas() actualiza la estructura de datos responsable de las Cajas con la Caja determinada por una NCta que se pasa por parámetro.

Figura 8.1: Diseño de las interfaces con módulos 2MIL, donde Banco, Caja y Cajas son clases en un diseño orientado a objetos.

Aún asi, definimos todas las operaciones (ver módulos 2MIL del diseño

8.1) para mostrar cómo queda integrada con el resto de la especificación. Es decir, la definición de la extracción esta compuesta por otras operaciones Z más primitivas, de las cuales algunas deben ser testeadas (ver Sección 4.1) en algún orden (ver orden sugerido en Sección 5.1). Esas operaciones primitivas pueden ser parte de la definición de otra operación que posiblemente esté al mismo nivel de abstracción que la operación de extracción. Es ahí cuando la integración se hace conveniente, porque si por algún motivo las operaciones primitivas ya fueron testeadas, cuando vamos a testear la extracción[2] entonces no tiene sentido testearlas nuevamente, y esto se logra usando los teoremas del Capítulo 6.

Los clientes se identifican por un único número que es el DNI y la información que guardan es su nombre y domicilio. Como no nos interesa expresar más estructura con esos tipos, serán representados como tipos básicos. De la misma manera hacemos con los número de cuenta.

$$[DNI, NOMBRE, DOMICILIO, NUMCTA]$$

El banco es la entidad que mantiene las estructuras generales para las operaciones que son los clientes, las cajas y los titulares. Tanto los tipos como el banco están idénticamente definidos como en la la Sección 3.

$$DINERO == \mathbb{Z}$$
$$HIS == \text{seq}\, DINERO$$
$$CajaAhorros == DINERO \times HIS$$
$$Cliente == NOMBRE \times DOMICILIO$$

Banco

$$cajas : NUMCTA \pfun CajaAhorros$$
$$clientes : DNI \pfun Cliente$$
$$titulares : DNI \pfun NUMCTA$$

Definimos algunos mensajes que lanzan las operaciones para los usuarios.

$$MENSAJES ::= ok | clienteExistente | saldoInsuficiente | montoNulo |$$
$$clienteInexistente | saldoNoNulo | montoSuperaMaxPermitido |$$
$$noEsTitular | cuentaExistente | cuentaInexistente$$

También definimos un monto máximo que pueden albergar las cajas.

[2]Por seguir el orden de la Sección 5.1 o porque ya se testearon otras operaciones que las usan.

$$montoCajaMax : DINERO$$

$$montoCajaMax = 50000$$

Definimos un estado inicial para el sistema.

BancoInicial

$$Banco$$

$$cajas = \emptyset$$
$$clientes = \emptyset$$
$$titulares = \emptyset$$

La operación *Extraer* se define de la siguiente manera:

$$Extraer == Entrada \land CajaAhorrosABanco \land CAExtraer$$

es la conjunción de *Entrada*, que es la validación del cliente interesado y del monto a extraer; *CAExtraer* que es la extracción efectiva del monto en la caja seleccionada por el cliente; y *CajaAhorrosABanco* que es la actualización de la extracción en el banco. La validación del monto es simplemente verificar que el monto sea positivo

$$ValidarMontoOk == [\Delta Banco;\ m? : DINERO;\ rep! : MENSAJES | m? > 0;\ rep! = ok]$$

$$ValidarMontoE == [\Xi Banco;\ m? : DINERO;\ rep! : MENSAJES | m? \leq 0;\ rep! = montoNulo]$$

$$ValidarMonto == ValidarMontoOk \lor ValidarMontoE$$

$$ValidarCliente == ValidarClienteOk \lor ValidarClienteE1 \lor ValidarClienteE2 \lor ValidarClienteE3$$

$$Entrada == ValidarCliente \land ValidarMonto$$

La validación del cliente tiene cuatro posibilidades. Que el *DNI* ingresado sea de un cliente existente y que la cuenta ingresada pertenezca al mismo,

ValidarClienteOk

$$\Delta Banco$$
$$caja, caja' : (DINERO \times HIS)$$
$$ncta? : NUMCTA$$
$$dni? : DNI$$

$$dni? \in \text{dom } clientes$$
$$ncta? \in \text{dom } cajas$$
$$titulares\ dni? = ncta?$$
$$cajas\ ncta? = caja$$

que el *DNI* no pertenezca a ningún cliente,

*ValidarClienteE*1

$$\Xi Banco$$
$$caja, caja' : (DINERO \times HIS)$$
$$ncta? : NUMCTA$$
$$dni? : DNI$$
$$rep! : MENSAJES$$

$$dni? \notin \text{dom } clientes$$
$$rep! = clienteInexistente$$

que el número de cuenta no pertenezca a ninguna caja de ahorros existente,

*ValidarClienteE*2

$$\Xi Banco$$
$$caja, caja' : (DINERO \times HIS)$$
$$ncta? : NUMCTA$$
$$dni? : DNI$$
$$rep! : MENSAJES$$

$$ncta? \notin \text{dom } cajas$$
$$rep! = cuentaInexistente$$

y que el titular no posea la caja ingresada.

*ValidarClienteE*3

$$\Xi Banco$$
$$caja, caja' : (DINERO \times HIS)$$
$$ncta? : NUMCTA$$
$$dni? : DNI$$
$$rep! : MENSAJES$$

$$titulares\ dni? \neq ncta?$$
$$rep! = noEsTitular$$

La extracción de la caja,

$$CAExtraer == CAExtraerOk \lor CAExtraerE$$

puede ser efectuada cuando el monto es menor o igual a lo que guarda la caja,

$$
\begin{array}{|l}
\hline
CAExtraerOk \\
caja, caja' : (DINERO \times HIS) \\
m? : DINERO \\
rep! : MENSAJES \\
\hline
m? \leq caja.1 \\
caja'.1 = caja.1 - m? \\
caja'.2 = caja.2 \frown \langle -m? \rangle \\
rep! = ok \\
\hline
\end{array}
$$

pero no puede efectuarse cuando es mayor.

$$
\begin{array}{|l}
\hline
CAExtraerE \\
caja, caja' : (DINERO \times HIS) \\
m? : \mathbb{Z} \\
rep! : MENSAJES \\
\hline
m? > caja.1 \\
caja' = caja' \\
rep! = saldoInsuficiente \\
\hline
\end{array}
$$

La actualización de la estructura *cajas* en el banco, por una modificación en una de sus cajas, se define simplemente como el reemplazo de la caja por la misma luego de su modificación. A diferencia del ejemplo del Capítulo 3, *CajaAhorrosABanco* es la especificación correspondiente a una subrutina del diseño.

$$
\begin{array}{|l}
\hline
CajaAhorrosABanco \\
\Delta Banco \\
caja, caja' : (DINERO \times HIS) \\
ncta? : NUMCTA \\
\hline
cajas' = cajas \oplus \{ncta? \mapsto caja'\} \\
clientes' = clientes \\
titulares' = titulares \\
\hline
\end{array}
$$

El depósito es muy parecido a la extracción. Justamente la idea es reutilizar esquemas de operación más primitivos. La única diferencia es que para depositar en una caja no hace falta ser el propietario de la misma, solo basta con que el monto no sea negativo.

$$Depositar == ValidarMonto \land CajaAhorrosABanco \land CADepositarOk$$

Y que además el monto final no supere el monto máximo permitido por el banco para las cajas.

CADepositarOk

$caja, caja' : (DINERO \times HIS)$
$m? : DINERO$
$rep! : MENSAJES$

$m? + caja1.1 \leq montoCajaMax$
$caja'.1 = caja.1 + m?$
$caja'.2 = caja.2 \frown \langle m? \rangle$
$rep! = ok$

Para la consulta y la baja de una caja se necesita validar al cliente.

$PedirSaldo == ValidarCliente \lor PedirSaldoOk$

$CerrarCaja == ValidarCliente \lor CerrarCajaOk$

PedirSaldoOk

$\Xi Banco$
$dni? : DNI$
$ncta? : NUMCTA$
$saldo! : DINERO$
$rep! : MENSAJES$

$saldo! = (cajas\ ncta?).1$
$rep! = ok$

CerrarCajaOk

$\Delta Banco$
$ncta? : NUMCTA$
$dni? : DNI$
$rep! : MENSAJES$

$cajas' = \{ncta?\} \lhd cajas$
$titulares' = titulares$
$clientes' = clientes$
$rep! = ok$

Por último, para dar de alta un nuevo cliente, no se incluye ninguna operación anterior y se opera siempre a nivel del banco.

$$NuevoCliente == NuevoClienteOk \lor NuevoClienteE$$

NuevoClienteOk

$\Delta Banco$
$dni? : DNI$
$nom? : NOMBRE$
$dom? : DOMICILIO$
$rep! : MENSAJES$

$dni? \notin \text{dom } clientes$
$clientes' = clientes \cup \{dni? \mapsto (nom?, dom?)\}$
$titulares' = titulares$
$cajas' = cajas$
$rep! = ok$

NuevoClienteE

$\Xi Banco$
$dni? : DNI$
$nom? : NOMBRE$
$dom? : DOMICILIO$
$rep! : MENSAJES$

$dni? \in \text{dom } clientes$
$rep! = clienteExistente$

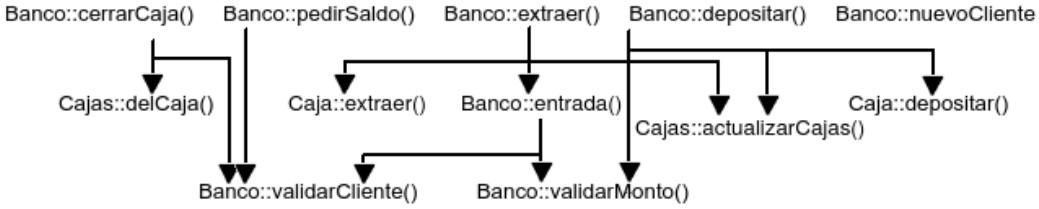

Figura 8.2: Diagrama $\overline{uses}$ para todas las subrutinas correspondientes a las operaciones especificadas.

En la Figura 8.2 se puede ver el esquema de $\overline{uses}$ para toda la especificación. Donde:
Depositar corresponde a la subrutina Banco :: depositar(),
CerrarCaja a Banco :: cerrarCaja(),
CADepositarOk a Caja :: depositar(),

Extraer a Banco :: extraer(),
CAExtraer a Caja :: extraer(),
Entrada a Banco :: entrada(),
PedirSaldo a Banco :: pedirSaldo(),
Nuevocliente a Banco :: nuevoCliente(),
CerrarCajaOk a Cajas :: delCajas(),
CajaAhorrosABanco a Cajas :: actualizarCajas(),
ValidarCliente a Banco :: validarCliente(),
y *ValidarMonto* a Banco :: validarMonto().

Este tipo de modularidad en la especificación, que sigue las recomendaciones de la Sección 4.3, nos permite graficar las ventajas de aplicar el testing de integración. El script para Fastest, el árbol de prueba con las clases y los casos están en el Apéndice A. Lo que se quiere mostrar a continuación es una comparación del TTF con integración con el TTF original para el mismo testing.

En la Figura 8.3 se pueden ver dos árboles de prueba generados para el esquema *Extraer*. El de la izquierda usando TTF tradicional y del de la derecha usando TTF de integración. Aplicar TTF con integración significa, generar casos para todos los esquemas que incluyen la definición de *Extraer* pero siguiendo el orden propuesto en la Sección 5.1, es decir para *ValidarCliente*, *ValidarMonto* (que en conjunción forman *Entrada*), luego para *Entrada*, *CajaAhorrosABanco* y *CAExtraer*. Al hacer esto Fastest automáticamente deja contraídos todos los esquemas seleccionados que forman parte de otro (ver Sección 5.2) y se genera un árbol por cada selección. Por último, para la generación de casos, Fastest toma cada caso generado para el esquema correspondiente y se fija si cumple el Teorema 1 con cada uno de los esquemas en conjunción (con que está su clase) recursivamente.

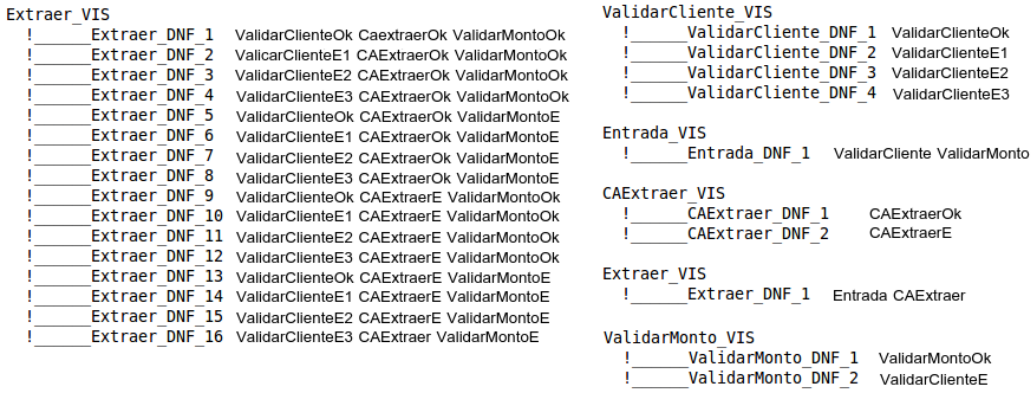

Figura 8.3: Árbol de prueba del TTF tradicional y con integración para *Extraer* generado con la táctica DNF.

Se puede ver que el árbol de prueba para el TTF tradicional tiene más clases de prueba, lo que es esperable dado que están todas las precondiciones expandidas y se aplica DNF a todos los predicados en conjunción. El inconveniente es que las precondiciones en *Extraer* quedan expandidas sin poder identificarse los esquemas a los que pertenecen. Esto dificulta saber qué subrutina estamos testeando y cómo lo estamos haciendo. Abajo se pueden ver las clases de pruebas a modo comparativo. A la izquierda el TTF tradicional y a la derecha con integración.

Extraer_DNF_1
Extraer_VIS

$dni? \in \text{dom } clientes$
$ncta? \in \text{dom } cajas$
$titulares\ dni? = ncta?$
$cajas\ ncta? = caja$
$m? > 0$
$m? \leq caja.1$

Extraer_DNF_1
Extraer_VIS

pre *Entrada*
pre *CAExtraer*

Por ejemplo, en la integración si queremos testear más exhaustivamente Banco :: entrada() aplicamos tácticas a *Entrada*, si queremos, una vez testeada esa subrutina, testear su relación con Caja :: extraer(), seleccionamos *Extraer* y quedará contraído *Entrada* y expandido *CAExtraer*. Si se quiere hacer de forma inversa, simplemente se hace lo inverso. El TTF tradicional deja al usuario toda la interpretación de la toda la lista de predicados. Resumiendo, el TTF de integración puede hacerse tan exhaustivo como el TTF tradicional

manejando a conveniencia la contracción y expansión.

A continuación mostramos la aplicación explícita del Teorema 1 para la operación *Extraer* a partir de correr el script en Fastest (ver Apéndice A) a toda la especificación del banco, donde se generan casos para todos los esquemas de operación definidos.

El caso generado desde *Extraer_VIS* es el que se muestra a continuación, el cual pasó el Teorema 1 para todos los esquemas incluidos, es decir *Entrada*, *CAExtraer* y *CajaAhorrosABanco*.

Extraer_DNF_1_TCASE

Extraer_DNF_1

$m? = -2147483648$
$titulares = \emptyset$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = \{2 \mapsto \langle 3 \rangle\}$

Notar que *Extraer_DNF_1_TCASE* para pasar el teorema tiene que cumplir con las siguientes clases de prueba 1. *Entrada* y 2. *CAExtraer*:

1.

Entrada_VIS

ValidarCliente_Decls
ValidarMonto_Decls

pre *ValidarCliente*
pre *ValidarMonto*

Entrada es un esquema sin predicados expandidos ya que ya fueron testeados *ValidarMonto* y *ValidarCliente*. En cambio, si alguno de los dos no hubiera sido testeado, entonces *Entrada* quedaría como se muestra en los dos siguientes esquemas. En el primero fue testeado *ValidarMonto* y no *ValidarCliente* y en el segundo a la inversa.

Entrada_VIS

$cajas : NUMCTA \pfun (DINERO \times HIS)$
$clientes : DNI \pfun (NOMBRE \times DOMICILIO)$
$titulares : DNI \pfun NUMCTA$
$caja : (DINERO \times HIS)$
$ncta? : NUMCTA$
$dni? : DNI$
ValidarMonto_Decls

$(dni? \in \mathrm{dom}\ clientes$
$ncta? \in \mathrm{dom}\ cajas$
$titulares\ dni? = ncta?$
$cajas\ ncta? = caja) \lor$
$dni? \notin \mathrm{dom}\ clientes \lor$
$ncta? \notin \mathrm{dom}\ cajas \lor$
$titulares\ dni? \neq ncta?$
pre *ValidarMonto*

Entrada_VIS

ValidarCliente_Decls
$cajas : NUMCTA \pfun (DINERO \times HIS)$
$clientes : DNI \pfun (NOMBRE \times DOMICILIO)$
$titulares : DNI \pfun NUMCTA$
$m? : DINERO$

---

pre *ValidarCliente*
$m? > 0 \lor m? \leq 0$

En ambos casos el *Extraer_DNF_1_TCASE* tiene que cumplir los predicados, pero dado que son totales, como es recomendado en la Sección 4.3, seguro tiene que pertenecer a alguna clase de prueba que se genere al aplicar DNF y cualquier táctica a partir de allí.

2. Luego *Extraer_DNF_1_TCASE* cumple además el Teorema 1 con *CAExtraer*, más específicamente con alguna de las clases de prueba que surja de la rama de *CAExtraerOk* como podría ser para el ejemplo *CAExtraer_DNF_1*.

CAExtraer_DNF_1

CAExtraer_VIS

$m? \leq caja.1$

Capítulo 9

Discusión

9.1 Recapitulación

Aunque nuestro interés específico es extender el TTF a testing de integración, los resultados mostrados podrían ser usados en otros lenguajes de especificación o métodos MBT. La mayoría de los resultados están basados en conceptos fundamentales de la Ingeniería de Software como la relación *uses*, lógica de primer orden y el MBT en general.

La descripción de operaciones como en (†) no es una restricción severa al uso del lenguaje y tiene un impacto no despreciable en la aplicación del TTF al testing de integración. La forma de (†) hace posible el cálculo automático de la relación *uses*, como mínimo en situaciones relevantes para el testing de integración. La organización del testing de integración alrededor de la relación *uses* da lugar a la optimización de estos procesos. Fue definido un orden en la jerarquía de subrutinas producida por la relación *uses*. Si la integración es producida siguiendo el orden sugerido, entonces muchos errores pueden ser capturados con una mínima cantidad de unidades ya integradas. Según cómo se defina la función $f(i)$ para la función de prioridad

$$pos(\mathsf{P}) = \sum_{i=0}^{N} \#(\mathfrak{D}(\mathsf{P}) \cap \mathfrak{U}_i) * f(i) \qquad (\ddagger)$$

proveerá un nivel más fino para guiar el testing de integración que tenga en cuenta no sólo la cantidad de subrutinas si no su nivel en la jerarquía. Todo esto apunta a hacer la búsqueda de la causa de un error sea más simple, descartando los errores tan pronto como sea posible.

Definir las relación *uses* no es sólo "un hito en el esfuerzo del diseño" [33], además es una tarea dificultosa raramente practicada por los desarrolladores de la industria. El hecho de que puede producir un impacto importante en reducir el costo del testing y además ser computada automáticamente desde una especificación Z, podría cambiar la relación costo-beneficio de Z y *uses*.

Testear una unidad en soledad es una declaración ambigua. En efecto, si P *uses* Q, ¿que significa testear P en soledad? Si significa no usar Q y en su reemplazo un *stub*, entonces el testing de unidad se enfrenta con el problema de la construcción de *stubs* correctos. Los *stubs* creados manualmente no sólo son propensos a errores sino que además son costosos [25, 4, 29]. El enfoque presentado también apunta a la reducción del costo de la generación de *stubs* y hacerlo de una manera segura para evitar la introducción de errores. Si la integración sigue la relación *uses* y cada unidad está certificada para al menos esas entradas usadas durante el testing, entonces pueden ser usadas como *stubs* de ellas mismas, siempre que sean llamadas como cuando fueron testeadas. Más aún, esos *stubs* implícitos de la relación *calls* pueden ser generados automáticamente, como se discutió en la Sección 4.3.

Los teoremas 1 y 2 dan condiciones simples bajo las cuales una subrutina puede ser usada como *stub* de ella misma, aunque sea algo probabilístico dado que la demostración depende de la hipótesis de uniformidad. Si tenemos al esquema B contraído en A y generamos un caso a para A, que se pueda generar desde a un caso b que cumpla alguna de las clases de prueba de B ya testeadas es algo probabilístico. Entonces una buena práctica que se puede agregar a la descriptas en la Sección 4.3 es que los esquemas de operaciones sean totales; generalmente esto quiere decir que tengan un esquema de error. De esta manera b terminará perteneciendo sí o sí a alguna de las clases generadas para B (ver Figura 6.2). En este caso estamos optando por eliminar el problema probabilístico especificando y testeando un poco más cada esquema, o por lo menos aquellos esquemas que forman parte de la definición de otros. De esta manera se está negociando el costo y riesgo de la creación de *stubs* por el costo de describir las relaciones *uses* y la aplicación de los teoremas 1 y 2, lo cual en muchos casos es automático—ver Sección 8. Finalmente, si estos teoremas no pueden ser probados para los casos de prueba generados desde A para P, significa que el caso no satisface ninguna hoja de B usada para testear Q, lo que es un indicio de que Q fue pobremente testeado porque ninguno de sus clientes debería llamarlo en una situación funcional que no fue cubierta durante el test.

El uso de subrutinas como *stubs* de ellas mismas, de alguna manera,

difumina la distinción entre testing de unidad e integración—tal vez $\mathfrak{U}_0$ es testing de unidad, y el resto de integración. Como sea, la integración deberá encontrar nuevos errores que son difíciles o imposibles de encontrar en el testing de unidad, tal como se mostró en la Sección 7. De hecho, el TTF extendido a testing de integración puede enfrentarse con casi cualquier error clasificado por Leung y White. El TTF y Z permiten análisis formales de algunas de estas clases de errores. El Teorema 3 y 4 y la táctica de testing MF muestran cómo nuestros resultados pueden ser extendidos más allá para manejar cuestiones particulares del testing de integración.

9.1.1 Trabajo relacionado

Hay mucho trabajo de investigación en el testing de integración, desde una perspectiva MBT [1, 35, 6, 3, 23, 25] o no [22, 16, 7, 32, 29, 28, 31], pero no es posible encontrar artículos que usen la teoría de diseño Parnas o la notación Z. Clements et al. [8, pages 68–71] prestan atención a la relación *uses* y remarcan su importancia en el testing de integración. En particular dicen que puede ser usado para acercar la búsqueda de la causa de un error encontrado durante el testing de integración pero no van más allá.

Leung y White [30, 31] estudian el testing de integración en el contexto del testing regresivo. Aunque usan la relación *calls*, ellos definen conjuntos de casos de prueba para testear subrutinas durante la integración que tienen algunas similitudes a lo presentado aquí. Aparentemente no está interesados en la generación de *stubs*, sino más bien en la reducción del número de test durante la regresión.

Benz [6] reconoce el hecho de que la relación crítica para el testing de integración no está modelada explícitamente y los métodos de MBT aplicados a la integración pueden cubrir grandes espacios de estados. En su trabajo usa modelos de tareas para especificar la interacción entre componentes. Ali et al. [3] usan diagramas de colaboración UML para modelar las interacciones entre clases y Statecharts para especificar el comportamiento. Proponen una lista de operadores de mutación que pueden ser usados para evaluar la efectividad de los métodos de integración. Como la lista apunta a programas orientados a objetos se prefirió aquí la taxonomía de errores propuesta por Leung y White, usada también por Orso [32]. Gallagher, Offutt y Cincotta usan Class State Machines (CSM) como método de especificación para las clases de los programas orientados a objetos. Las CSM son combinadas en un grafo de flujo (Component Flow Graph) el cual es usado para derivar el test de integración.

Schätz y Pfaller tienen como objetivo testear componentes que sólo pueden ser accedidos a traves de su interfaz. Usan un sistema de transiciones para

modelar el comportamiento de los componentes y un sistema jerárquico de transiciones para modelar las interacciones de los componentes. Definen la noción de Caso de Prueba de Integración Satisfecho (Satisfied Integration Test Case) el cual juega un rol similar a los Teoremas 1 y 2. Otro trabajo que se concentra en un problema específico, *carving and replay* basado en testing de integración, es el de Elbaum y sus colegas [22]. Como sea, los cuatro pasos del testing de unidad que usan son los mismos usados en Fastest: identificar un estado del programa, setearlo, ejecutarlo y evaluar los resultados. Además notan que un método podría depender solo de una pequeña parte del estado del programa y desarrollan recursos en el contexto del *carving and replay*.

Hartmann, Imoberdorf y Meisinger [25] usan métodos basados en particiones de categorías para generar casos de prueba desde especificaciones UML Statecharts que especifican el comportamiento cuyas interacciones son descriptas mediante conceptos tomados de CSP. Las particiones de categorías son lo que esencialmente hace TTF con el *VIS* de una operación Z. Apuntan al problema de la generación de *stubs* pero no está claro cómo sus métodos reducen el número de *stubs* generados a mano.

Labiche et al. [29] definen una estrategia de integración basada en diagramas de clases con el objetivo de minimizar la generación de *stubs*. Esencialmente testean las clases después de las clases de las cuales dependen. El orden de la integración de Labiche es una extensión de Kung [28] cuando están presentes las dependencias dinámicas y clases abstractas. Como sea, las clases o diagramas similares raramente incluyen especificación funcional de las clases. De hecho, estos métodos hacen un análisis sintáctico de esos diagramas lo que resulta en un número grande de dependencias ya que no solo incluyen la dependencia de clase "used" sino también "called".

9.2 Algunas cuestiones más específicas

En los teoremas 1 y 2, probar que $B_j^A(x) \neq \emptyset$ involucra tanto la evaluación de predicados Z constantes como la solución del problema de satisfacibilidad. En efecto, si $vars(B) \subseteq vars(A)$ entonces todas las variables libres en B_j serán remplazadas por valores constantes cuando se calcula $B_j^A(x)$; de otra manera, habrá variables libres en $B_j(x)$. En el primer caso $B_j^A(x) \neq \emptyset$ puede ser resuelto automáticamente; en el segundo caso es necesario decidir si $B_j^A(x)$ es satisfacible o no. Este problema es indecidible porque $B_j^A(x)$ puede ser un predicado de primer orden sobre la teoría de conjuntos. Como sea, Fastest usa técnicas avanzadas de Programación Lógica de Restricciones (la herramienta $\{log\}$) para resolver estos predicados con muy buenos resultados

para especificaciones reales [21, 11, 17]. Entonces, aún cuando $B_j^A(x)$ tiene variables libres los teoremas 1 y 2 pueden ser aplicados automáticamente en muchas situaciones (ver apéndice C). Además, un primer candidato para la substitución de estas variables libres son los valores constantes de los casos generados en B_j cuando Q fue testeado; aunque, en general, no será necesario satisfacer el predicado. De esta manera, el testing de integración puede reutilizar estos casos de prueba generados durante el testing de unidad. Resumiendo, este teorema puede ser automáticamente generado y aplicado en muchos casos. De hecho para el caso de estudio en la sección siguiente 8, automatizamos la aplicación del Teorema 1 en Fastest.

Estrictamente hablando, estos teoremas necesitan ser aplicados solo para los casos donde P llama a Q, y no para todos de ellos. Entonces, sería necesario tener un procedimiento automático para conocer qué casos de prueba harán que P llamen a Q.

Capítulo 10

Conclusiones y trabajo futuro

El TTF ha sido extendido al testing de integración agregando, en principio, una buena cobertura durante éste nivel de testing. El impacto favorable que *uses* tiene en el testing podría hacer que los desarrolladores efectivamente incorporen esta documentación, así se estaría reusando un documento clave en el diseño. Más aún, si una especificación lógica está estructurada inteligentemente, *uses* se puede generar automáticamente. La extensión minimiza la necesidad de generar manualmente *stubs*, sea dando condiciones simples que permieten que un *stub* pueda ser generado automáticamente o estableciendo directamente cuándo una subrutina puede ser usada como *stub* de ella misma.

Como sea, debería investigarse qué táctica de prueba debería ser aplicada a dos subrutinas que pertenecen a la relación *uses* para probar los Teoremas 1 y 2 para todas las clases de prueba, y además sigan proveyendo una buena cobertura en el testing de unidad para todas ellas. En particular, debería establecerse qué hacer si estos teoremas no pueden aplicarse a un caso de prueba, ya que estaría indicando un testeo pobre de la subrutina llamada. Asimismo, dado que estos teoremas deberían aplicarse solo a los casos de prueba que producen el llamado de una subrutina, sería importante saber cuándo un caso producirá tal llamada (se reduciría el número de veces que los teoremas deben aplicarse). Otra cuestión es estudiar la relación del operador Z θ y la promoción de operaciones con el testing de integración.

Apéndice A

Árbol y casos de prueba generados para la especificación completa presentada en el Capítulo 8

Presentamos el script (ver Figura A.1) para el caso del Capítulo 8 y el árbol completo luego de correrlo en Fastest.

Primero se carga la especificación (`loadsec banco.tex`), se seleccionan todas las operaciones (`selop CerrarCajaOk`, etc.) y se reemplazan las definiciones axiomáticas [9] en las operaciones (`replaceaxdef`). Luego se generan los árboles de pruebas para cada operación (`genalltt`) y se agregan las tácticas a en las hojas de los árboles (`ValidarMonto_DNF_2 SP \leq m? \leq 0`, etc.). Finalmente se actualizan los árboles (`genalltt`) y se generan los casos de prueba para los mismos (`genalltca`).

```
loadspec banco.tex
selop CerrarCajaOk
selop CajaAhorrosABanco
selop CADepositarOk
selop ValidarMonto
selop ValidarCliente
selop CAExtraer
selop Entrada
selop Extraer
selop Depositar
selop CerrarCaja
selop NuevoCliente
selop PedirSaldo
replaceaxdef
genalltt
addtactic ValidarMonto_DNF_2 SP \leq m? \leq 0
addtactic ValidarMonto_DNF_1 SP > m? > 0
addtactic CAExtraer_DNF_2 SP > m? > caja . 1
addtactic CAExtraer_DNF_2 SP \leq m? \leq caja . 1
addtactic CAExtraer_DNF_1 SP \leq m? \leq caja . 1
addtactic CADepositarOk_DNF_1 SP \leq m? \leq 50000
addtactic ValidarCliente_DNF_1 SP \in dni? \in \dom clientes
addtactic ValidarCliente_DNF_1 SP \in ncta? \in \dom cajas
addtactic ValidarCliente_DNF_1 SP = titulares~dni? = ncta?
addtactic ValidarCliente_DNF_2 SP \notin dni? \notin \dom clientes
addtactic ValidarCliente_DNF_3 SP \notin ncta? \notin \dom cajas
addtactic NuevoCliente_DNF_1 SP \notin dni? \notin \dom clientes
addtactic NuevoCliente_DNF_2 SP \in dni? \in \dom clientes
genalltt
genalltca
```

Figura A.1: Script para Fastest para todas las operaciones definidas en la especificación completa del *Banco*.

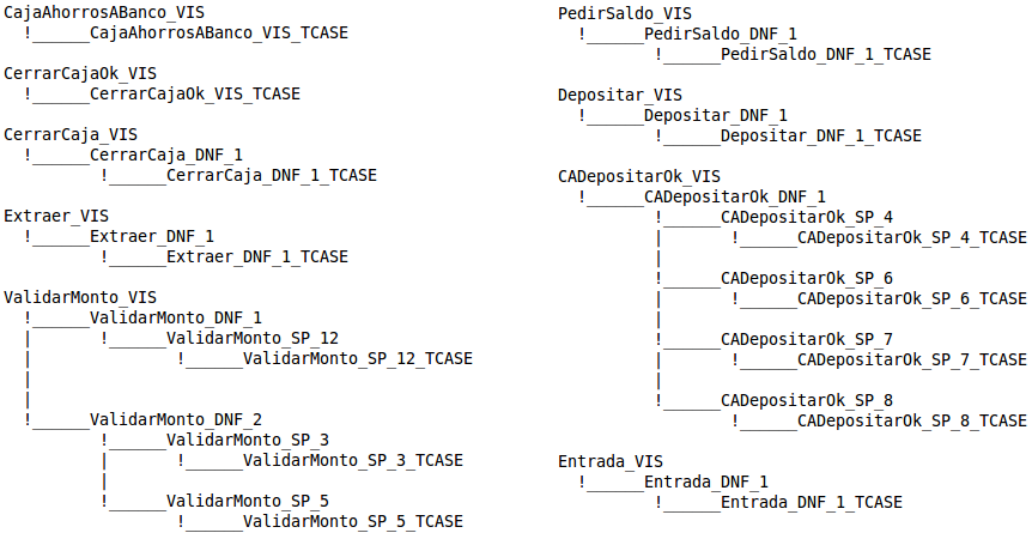

Figura A.2: Árboles para *CajaAhorrosABanco*, *CerrarOk*, *CerrarCaja*, *Extraer*, *ValidarMonto*, *PedirSaldo*, *Depositar*, *CADepositarOk*, *Entrada* generados por Fastest.

Figura A.3: Árboles para *ValidarCliente* y *CAExtraer*.

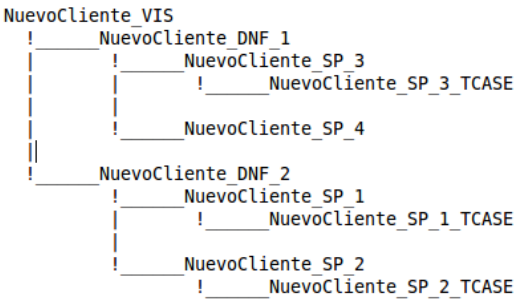

Figura A.4: Árbol para *NuevoCliente*.

Finalmente todos los casos generados por el script para Fastest (Figura A.1):

ValidarCliente_SP_10_TCASE

ValidarCliente_SP_10

$titulares = \{(dni1 \mapsto 0)\}$
$ncta? = 0$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3))\}$
$cajas = \{(0 \mapsto (6 \mapsto \langle 7 \rangle))\}$
$caja = (8 \mapsto \langle 9 \rangle)$

ValidarCliente_SP_11_TCASE

ValidarCliente_SP_11

$titulares = \{(dni1 \mapsto 1)\}$
$ncta? = 1$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3))\}$
$cajas = \{(1 \mapsto (6 \mapsto \langle 7 \rangle))\}$
$caja = (8 \mapsto \langle 9 \rangle)$

ValidarCliente_SP_13_TCASE

ValidarCliente_SP_13

$titulares = \{(dni1 \mapsto 0)\}$
$ncta? = 0$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3))\}$
$cajas = \{(0 \mapsto (9 \mapsto \langle 10 \rangle)),$
$\quad (6 \mapsto (7 \mapsto \langle 8 \rangle))\}$
$caja = (11 \mapsto \langle 12 \rangle)$

ValidarCliente_SP_14_TCASE

ValidarCliente_SP_14

$titulares = \{(dni1 \mapsto 1)\}$
$ncta? = 1$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3))\}$
$cajas = \{(1 \mapsto (9 \mapsto \langle 10 \rangle)),$
$\quad (6 \mapsto (7 \mapsto \langle 8 \rangle))\}$
$caja = (11 \mapsto \langle 12 \rangle)$

ValidarCliente_SP_18_TCASE

ValidarCliente_SP_18

$titulares = \{(dni1 \mapsto 0)\}$
$ncta? = 0$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3)),$
$\quad (dni4 \mapsto (nombre5 \mapsto domicilio6))\}$
$cajas = \{(0 \mapsto (9 \mapsto \langle 10 \rangle))\}$
$caja = (11 \mapsto \langle 12 \rangle)$

ValidarCliente_SP_19_TCASE

ValidarCliente_SP_19

$titulares = \{(dni1 \mapsto 1)\}$
$ncta? = 1$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3)),$
$\quad (dni4 \mapsto (nombre5 \mapsto domicilio6))\}$
$cajas = \{(1 \mapsto (9 \mapsto \langle 10 \rangle))\}$
$caja = (11 \mapsto \langle 12 \rangle)$

ValidarCliente_SP_21_TCASE

ValidarCliente_SP_21

$titulares = \{(dni1 \mapsto 0)\}$
$ncta? = 0$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3)),$
$\quad (dni4 \mapsto (nombre5 \mapsto domicilio6))\}$
$cajas = \{(0 \mapsto (12 \mapsto \langle 13 \rangle)),$
$\quad (9 \mapsto (10 \mapsto \langle 11 \rangle))\}$
$caja = (14 \mapsto \langle 15 \rangle)$

ValidarCliente_SP_22_TCASE

ValidarCliente_SP_22

$titulares = \{(dni1 \mapsto 1)\}$
$ncta? = 1$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3)),$
$\quad (dni4 \mapsto (nombre5 \mapsto domicilio6))\}$
$cajas = \{(1 \mapsto (12 \mapsto \langle 13 \rangle)),$
$\quad (9 \mapsto (10 \mapsto \langle 11 \rangle))\}$
$caja = (14 \mapsto \langle 15 \rangle)$

ValidarCliente_SP_3_TCASE

ValidarCliente_SP_3

$titulares = \emptyset$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = (2 \mapsto \langle 3 \rangle)$

ValidarCliente_SP_1_TCASE

ValidarCliente_SP_1

$titulares = \emptyset$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = (2 \mapsto \langle 3 \rangle)$

ValidarCliente_DNF_4_TCASE

ValidarCliente_DNF_4

$titulares = \{(dni1 \mapsto 1)\}$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = (2 \mapsto \langle 3 \rangle)$

NuevoCliente_SP_3_TCASE

NuevoCliente_SP_3

$dom? = domicilio1$
$titulares = \emptyset$
$nom? = nombre3$
$dni? = dni2$
$clientes = \emptyset$
$cajas = \emptyset$

NuevoCliente_SP_1_TCASE

NuevoCliente_SP_1

$dom? = domicilio4$
$titulares = \emptyset$
$nom? = nombre5$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3))\}$
$cajas = \emptyset$

NuevoCliente_SP_2_TCASE

NuevoCliente_SP_2

$dom? = domicilio7$
$titulares = \emptyset$
$nom? = nombre8$
$dni? = dni1$
$clientes = \{(dni1 \mapsto (nombre2 \mapsto domicilio3)),$
$\quad (dni4 \mapsto (nombre5 \mapsto domicilio6))\}$
$cajas = \emptyset$

PedirSaldo_DNF_1_TCASE

PedirSaldo_DNF_1

$titulares = \emptyset$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = (2 \mapsto \langle 3 \rangle)$

Depositar_DNF_1_TCASE

Depositar_DNF_1

$m? = \overline{\ }2147483648$
$titulares = \emptyset$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = (2 \mapsto \langle 3 \rangle)$

CADepositarOk_SP_4_TCASE

CADepositarOk_SP_4

$m? = \overline{\ }2147483648$
$caja = (1 \mapsto \langle 2 \rangle)$

CADepositarOk_SP_6_TCASE

CADepositarOk_SP_6

$m? = 0$

$caja = (1 \mapsto \langle 2 \rangle)$

CADepositarOk_SP_7_TCASE

CADepositarOk_SP_7

$m? = 1$

$caja = (1 \mapsto \langle 2 \rangle)$

CADepositarOk_SP_8_TCASE

CADepositarOk_SP_8

$m? = 50000$

$caja = (1 \mapsto \langle 2 \rangle)$

Entrada_DNF_1_TCASE

Entrada_DNF_1

$m? = \overline{\ }2147483648$

$titulares = \emptyset$

$ncta? = 0$

$dni? = dni1$

$clientes = \emptyset$

$cajas = \emptyset$

$caja = (2 \mapsto \langle 3 \rangle)$

CAExtraer_SP_46_TCASE

CAExtraer_SP_46

$m? = \overline{\ }2147483648$

$caja = (\overline{\ }2147483647 \mapsto \langle 1 \rangle)$

CAExtraer_SP_47_TCASE

CAExtraer_SP_47

$m? = \overline{\ }2147483648$

$caja = (\overline{\ }2147483648 \mapsto \langle 1 \rangle)$

CAExtraer_SP_48_TCASE

CAExtraer_SP_48

$m? = \bar{}2147483648$

$caja = (0 \mapsto \langle 1 \rangle)$

CAExtraer_SP_49_TCASE

CAExtraer_SP_49

$m? = \bar{}2147483648$

$caja = (1 \mapsto \langle 1 \rangle)$

CAExtraer_SP_50_TCASE

CAExtraer_SP_50

$m? = 0$

$caja = (0 \mapsto \langle 1 \rangle)$

CAExtraer_SP_51_TCASE

CAExtraer_SP_51

$m? = 0$

$caja = (1 \mapsto \langle 1 \rangle)$

CAExtraer_SP_52_TCASE

CAExtraer_SP_52

$m? = 1$

$caja = (2 \mapsto \langle 1 \rangle)$

CAExtraer_SP_53_TCASE

CAExtraer_SP_53

$m? = 1$

$caja = (1 \mapsto \langle 1 \rangle)$

CAExtraer_SP_3_TCASE

CAExtraer_SP_3

$m? = 0$

$caja = (\overline{\ }2147483648 \mapsto \langle 1 \rangle)$

CAExtraer_SP_4_TCASE

CAExtraer_SP_4

$m? = 1$

$caja = (0 \mapsto \langle 1 \rangle)$

CajaAhorrosABanco_VIS_TCASE

CajaAhorrosABanco_VIS

$titulares = \emptyset$

$ncta? = 0$

$clientes = \emptyset$

$cajas = \emptyset$

$caja = (1 \mapsto \langle 2 \rangle)$

CerrarCajaOk_VIS_TCASE

CerrarCajaOk_VIS

$titulares = \emptyset$

$ncta? = 0$

$dni? = dni1$

$clientes = \emptyset$

$cajas = \emptyset$

CerrarCaja_DNF_1_TCASE

CerrarCaja_DNF_1

$titulares = \emptyset$

$ncta? = 0$

$dni? = dni1$

$clientes = \emptyset$

$cajas = \emptyset$

$caja = (2 \mapsto \langle 3 \rangle)$

Extraer_DNF_1_TCASE

Extraer_DNF_1

$m? = \overline{\ }2147483648$
$titulares = \emptyset$
$ncta? = 0$
$dni? = dni1$
$clientes = \emptyset$
$cajas = \emptyset$
$caja = (2 \mapsto \langle 3 \rangle)$

ValidarMonto_SP_12_TCASE

ValidarMonto_SP_12

$m? = 1$
$titulares = \emptyset$
$clientes = \emptyset$
$cajas = \emptyset$

ValidarMonto_SP_3_TCASE

ValidarMonto_SP_3

$m? = \overline{\ }2147483648$
$titulares = \emptyset$
$clientes = \emptyset$
$cajas = \emptyset$

ValidarMonto_SP_5_TCASE

ValidarMonto_SP_5

$m? = 0$
$titulares = \emptyset$
$clientes = \emptyset$
$cajas = \emptyset$

Apéndice B

Contracción de los esquemas de operación

Como se explicó en la Sección 5.2, la no expansión de los esquemas de operaciones que ya fueron probados, además de ser una característica teórica del testing de integración por representar la capacidad progresiva, modular y ordenada del testing, es una necesidad práctica a la hora de usar el TTF. Las especificaciones reales contarán con una cantidad y tamaño no despreciable de operaciones, que seguramente estarán anidadas de maneras complejas, por lo cual la sola aplicación del TTF tradicional y la correspondiente interpretación de lo que se está probando se tornará costosa.

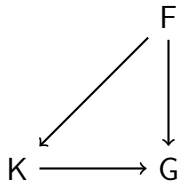

Figura B.1: Relaciones de inclusión de esquemas de operaciones Z.

El orden propuesto en la Sección 5 para la Figura B.1 indica debemos testear G, luego K, luego F. Definimos entonces una especificación simple para la misma y mostraremos, con ayuda de la herramienta Fastest, como queda la inclusión de esquemas siguiendo varios órdenes para el testing, a modo de esclarecer como van quedando incluidos a medida que se va trabajando (sea o no el orden propuesto).

Sea la siguiente especificación para la Figura B.1.

$G == [g, g' : \mathbb{Z} | g > 1]$
$K == G \vee [k, k' : \mathbb{Z} | k > 1]$
$F == G \vee K$

Algunas de las situaciones posibles son[1]:
1) Se probó G y K, y se quiere probar F.
2) Se probó G y se quiere probar F.
3) Se probó K y se quiere probar F.

Los comandos de Fastest para 1), 2) y 3):

1)	2)	3)
`selop F`	`selop F`	`selop F`
`selop K`	`selop G`	`selop K`
`selop G`	`genalltt`	`genalltt`
`genalltt`		

Y se obtienen las siguientes clases de prueba respectivamente:

1)
$G_1^{DNF} == [G^{VIS} | g > 1]$
$K_1^{DNF} == [K^{VIS} | \text{pre } G]$
$K_2^{DNF} == [K^{VIS} | k > 1]$
$F_1^{DNF} == [F^{VIS} | \text{pre } G]$
$F_2^{DNF} == [F^{VIS} | \text{pre } K]$

2)
$G_1^{DNF} == [G^{VIS} | g > 1]$
$F_1^{DNF} == [F^{VIS} | \text{pre } G]$
$F_2^{DNF} == [F^{VIS} | k > 1]$

3)
$K_1^{DNF} == [K^{VIS} | g > 1]$
$K_2^{DNF} == [K^{VIS} | k > 1]$
$F_1^{DNF} == [F^{VIS} | g > 1]$
$F_2^{DNF} == [F^{VIS} | \text{pre } K]$

En 1) están todos los esquemas contraídos, solamente van a tener precondiciones las variables que no vienen desde un esquema incluido. Como G y K permanecen contraídas en F^{DNF} las variables g y k no tienen restricciones en el esquema, no así g en G^{DNF} y k en K^{DNF}. En 2) La variable k mantiene la precondición en F_2^{DNF} porque corresponde a la inclusión de K^{VIS}, correspondiente a la subrutina K, la cual no fue testeada. En F_1^{DNF}

[1]En cada caso solo se aplica la táctica DNF porque únicamente se quiere mostrar el estado de la inclusión de esquemas.

las dos variables están sin precondiciones porque provienen de la inclusión de G^{VIS}, correspondiente a G, que fue probada. Por último en 3) cómo G no fue testeada, las dos variables (g y k) permanecen con sus precondiciones en K^{DNF}. En F_1^{DNF} está expandido solamente G^{VIS} y queda k sin precondición, porque ésta sólo puede provenir de K^{VIS} que fue testeada y está contraída. En F_2^{DNF} quedan las dos variables sin precondiciones porque provienen de la precondición de K^{VIS}.

Cabe decir que otras direcciones en las flechas de la Figura B.1 no producen un cambio relevante para analizar, quedan modelos equivalentes, salvo el que produce un ciclo, el cual representa una especificación errónea.

Apéndice C

Aplicación automática del Teorema 1 con Fastest

Vamos a mostrar el procedimiento de la aplicación del teorema con el siguiente ejemplo:

$$C == [x, x' : \mathbb{Z} | x > 10 \land x < 20]$$
$$B == [x, x' : \mathbb{Z} | x > 15 \land x < 25] \land C$$
$$A == [x, x', y : \mathbb{Z} | x > 18 \land x < 26 \land y > 0] \land B$$

Queremos generar un caso para cada esquema de operación de manera de producir el testing de integración como indica el Capítulo 6, es decir, dejando contraídos los esquemas ya testeados en los esquemas compuestos. Con la Fastest se logra por ejemplo corriendo el siguiente script: `selopall; genalltt; genalltca`.

La generación desde C se logra sin ningún detalle que notar. Para la generación desde B se logra como tradicionalmente se hacia en el TTF, salvo que en esta oportunidad C queda contraído en la operación B, por lo que las precondiciones de C no aparecen en B, entonces para que el caso de B sirva tendrá que pasar el Teorema 1 con C. Para la generación desde A se debe aplicar el Teorema 1 con B, que a su vez deberá hacer lo propio con C. O sea, para el caso de A se aplica el teorema recursivamente dos veces.

Internamente lo que hace Fastest es, por cada caso generado, conjugarlo con cada esquema contraído y ver si satisface el Teorema, eso lo hace recursivamente. Por ejemplo, para la generación desde A, supongamos que se genera el caso $a = [x, y : \mathbb{Z} | x = 19 \land y = 1]$ se conjuga entonces con B generando el esquema (o clase de prueba) temporal $aB = [x, x', y : \mathbb{Z} | x =$

$19 \wedge y = 1 \wedge x > 15 \wedge x < 25] \wedge C$.

Luego se resuelve aB [1] quedando el nuevo caso $b = [x : \mathbb{Z} | x = 19]$ que a su vez tendrá que ser conjugado de idéntica manera con C. Notar que la variable y no debería estar en aB porque no forma parte de B. No debería formar parte de la comprobación del Teorema, pero se lo hace así por razones de eficiencia, no modifica en nada la comprobación del Teorema. El siguiente pseudo algoritmo lo describe el proceso para el caso general.

```
genrarCaso(clase){
    //el motor de Fastest genera el caso con resolver
    caso = resolver(clase);
    //integrar aplica el Teorema 1 con el caso generado
    return integrar(caso,clase);
}
integrar(caso,clase){
    para todos los esquemas contraidos (op_i) en clase{
        // genera la clase temporal,
        nuevaClase = hacerClase(caso,op_i);
        //paso recursivo, chequeamos
        //el teorema con el caso generado,
        nuevoCaso = generarCaso(nuevaClase);
        if (!nuevoCaso) //fallo el teorema 1;
                return null;
        }
    return caso; //pas'o el teorema
}
```

Veamos el algoritmo guiados por el ejemplo, primero `resolver` genera el caso a, que se pasa por parámetro conto con A a `integrar`. Allí `hacerClase` es la encargada de generar la clase temporal del ejemplo aB, ya que el único esquema contraído en A es B (`Op_i`). El siguiente paso es la llamada recursiva a `generarCaso` con el esquema aB. Allí primero `resolver` genera el caso b, finalmente en `integrar` se tratará de conjugar a b con todos los esquemas contraídos en aB, es decir con C.

[1] Con el mismo motor con que se generan los casos de prueba.

Si aplicamos al ejemplo los siguientes comandos para Fastest, `selopall;genalltt;genalltca` sucede lo siguiente.

A_VIS
    !____A_DNF_1
        !____A_DNF_1_TCASE
B_VIS
    !____B_DNF_1
        !____B_DNF_1_TCASE
C_VIS
    !____C_DNF_1
        !____C_DNF_1_TCASE

$A\_DNF\_1 == [A\_VIS | x > 18 \land x < 26 \land y > 0 \land \text{pre } B]$
$A\_DNF\_1\_TCASE == [A\_DNF\_1 | x = 19 \land y = 1]$

$B\_DNF\_1 == [B\_VIS | x > 15 \land x < 25 \land \text{pre } C]$
$B\_DNF\_1\_TCASE == [B\_DNF\_1 | x = 16]$

$C\_DNF\_1 == [C\_VIS | x > 10 \land x < 20]$
$C\_DNF\_1\_TCASE == [C\_DNF\_1 | x = 11]$

El caso interesante surge si cambiamos A por $A_2 == [x, x', y : \mathbb{Z} | x > 21 \land x < 26 \land y > 0] \land B$. Este esquema generará un caso que cumple el Teorema 1 con B pero no el paso reursivo con C. Fastest genera el caso pero con un mensaje de advertencia.

A_VIS
    !____A_DNF_1
        !____A_DNF_1_TCASE not integrated with B;
B_VIS
    !____B_DNF_1
        !____B_DNF_1_TCASE
C_VIS
    !____C_DNF_1
        !____C_DNF_1_TCASE

Donde $A\_DNF\_1\_TCASE == [A\_DNF\_1 | x = 22 \land y = 1]$.

Bibliografía

[1] Aiguier, M., Boulanger, F., Kanso, B.: A formal abstract framework for modelling and testing complex software systems. Theor. Comput. Sci. 455, 66–97 (Oct 2012), http://dx.doi.org/10.1016/j.tcs.2011.12.072

[2] Alexander, R.T., Offutt, A.J.: Criteria for testing polymorphic relationships. In: Proceedings of the 11th International Symposium on Software Reliability Engineering. pp. 15–. ISSRE '00, IEEE Computer Society, Washington, DC, USA (2000), http://dl.acm.org/citation.cfm?id=851024.856208

[3] Ali, S., Briand, L.C., Rehman, M.J.u., Asghar, H., Iqbal, M.Z.Z., Nadeem, A.: A state-based approach to integration testing based on uml models. Inf. Softw. Technol. 49(11-12), 1087–1106 (Nov 2007), http://dx.doi.org/10.1016/j.infsof.2006.11.002

[4] Baresi, L., Pezzè, M.: An introduction to software testing. Electron. Notes Theor. Comput. Sci. 148(1), 89–111 (Feb 2006), http://dx.doi.org/10.1016/j.entcs.2005.12.014

[5] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edn. (2003)

[6] Benz, S.: Combining test case generation for component and integration testing. In: Proceedings of the 3rd international workshop on Advances in model-based testing. pp. 23–33. A-MOST '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1291535.1291538

[7] Buy, U., Orso, A., Pezze, M.: Automated testing of classes. In: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis. pp. 39–48. ISSTA '00, ACM, New York, NY, USA (2000), http://doi.acm.org/10.1145/347324.348870

[8] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)

[9] Cristiá, Maximiliano and Rodríguez Monetti, Pablo and Albertengo, Pablo: The Fastest 1.3.6 User's Guide, CIFASIS (2010). url = http://www.fceia.unr.edu.ar/ mcristia

[10] Cristiá, M.: Una Teoría para el Diseño de Software (2010), http://www.fceia.unr.edu.ar/ingsoft/intro-diseno.pdf

[11] Cristiá, M., Albertengo, P., Frydman, C., Plüss, B., Monetti, P.R.: Tool support for the Test Template Framework. Software Testing, Verification and Reliability pp. n/a–n/a (2012), http://dx.doi.org/10.1002/stvr.1477

[12] Cristiá, M., Albertengo, P., Rodríguez Monetti, P.: Pruning testing trees in the Test Template Framework by detecting mathematical contradictions. In: Fiadeiro, J.L., Gnesi, S. (eds.) SEFM. pp. 268–277. IEEE Computer Society (2010)

[13] Cristiá: Introducción a la notación Z. Facultad de Ciencias Exactas, Ingeniería y Agrimensura Universidad Nacional de Rosario (2012). http://www.fceia.unr.edu.ar/asist/

[14] Cristiá, M., Joaquín C.: Definición de estrategias para la aplicación automática de tácticas de testing en el marco del TTF y Fastest. Facultad de Ciencias Exactas, Ingeniería y Agrimensura Universidad Nacional de Rosario (2014).

[15] Cristiá, M., Frydman C., Mesuro J.: Integration Testing in the Test Template Framework. In: 17th International Conference on Fundamental Approaches to Software Engineering (FASE), ETAPS 2014: 5-13 April 2014, Grenoble, France. http://link.springer.com/chapter/10.1007/978-3-642-54804-8_28

[16] Alexander, R.T., Offutt, A.J.: Criteria for testing polymorphic relationships. In: Proceedings of the 11th International Symposium on Software Reliability Engineering. pp. 15–. ISSRE '00, IEEE Computer Society, Washington, DC, USA (2000), http://dl.acm.org/citation.cfm?id=851024.856208

[17] Frydman, C.: Applying SMT solvers to the Test Template Framework. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings 7th Workshop on Model-Based Testing, Tallinn, Estonia, 25 March 2012. Electronic Proceedings in Theoretical Computer Science, vol. 80, pp. 28–42. Open Publishing Association (2012)

[18] Cristiá, M., Frydman, C.S.: Extending the Test Template Framework to deal with axiomatic descriptions, quantifiers and set comprehensions. In: Derrick, J., Fitzgerald, J.A., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ. Lecture Notes in Computer Science, vol. 7316, pp. 280–293. Springer (2012)

[19] Cristiá, M., Hollmann, D., Albertengo, P., Frydman, C.S., Monetti, P.R.: A language for test case refinement in the Test Template Framework. In: Qin, S., Qiu, Z. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 6991, pp. 601–616. Springer (2011)

[20] Cristiá, M., Rodríguez Monetti, P.: Implementing and applying the Stocks-Carrington framework for model-based testing. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM. Lecture Notes in Computer Science, vol. 5885, pp. 167–185. Springer (2009)

[21] Cristiá, M., Rossi, G., Frydman, C.S.: {log} as a test case generator for the Test Template Framework. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM. Lecture Notes in Computer Science, vol. 8137, pp. 229–243. Springer (2013)

[22] Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 253–264. SIGSOFT '06/FSE-14, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1181775.1181806

[23] Gallagher, L., Offutt, J., Cincotta, A.: Integration testing of object-oriented components using finite state machines: Research articles. Softw. Test. Verif. Reliab. 16(4), 215–266 (Dec 2006), http://dx.doi.org/10.1002/stvr.v16:4

[24] Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of software engineering (2nd ed.). Prentice Hall (2003)

[25] Hartmann, J., Imoberdorf, C., Meisinger, M.: UML-based integration testing. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 60–70. ISSTA '00, ACM, New York, NY, USA (2000), http://doi.acm.org/10.1145/347324.348872

[26] Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. 41(2), 1–76 (2009)

[27] Jacky, J.: The way of Z: practical programming with formal methods. Cambridge University Press, New York, NY, USA (1996)

[28] Kung, D.C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y.: Class firewall, test order, and regression testing of object-oriented programs. JOOP 8(2), 51–65 (1995)

[29] Labiche, Y., Thévenod-Fosse, P., Waeselynck, H., Durand, M.H.: Testing levels for object-oriented software. In: Proceedings of the 22nd International Conference on Software Engineering. pp. 136–145. ICSE '00, ACM, New York, NY, USA (2000), http://doi.acm.org/10.1145/337180.337197

[30] Leung, H.K.N., White, L.: Insights into testing and regression testing global variables. Journal of Software Maintenance 2(4), 209–222 (1990)

[31] Leung, H.K.N., White, L.: A study of integration testing and software regression at the integration level. In: Conference on Software Maintenance-90. pp. 290–301. San Diego, CA (1990)

[32] Orso, A.: Integration Testing of Object-Oriented Software. Ph.D. thesis, Politecnico di Milano, Milan, Italy (february 1999)

[33] Parnas, D.L.: Designing software for ease of extension and contraction. In: ICSE '78: Proceedings of the 3rd international conference on Software engineering. pp. 264–277. IEEE Press, Piscataway, NJ, USA (1978)

[34] Parnas, D.L.: On the criteria to be used in descomposing system into modules. In: Comunications of the ACM, vol. 15, no. 12, pp. 1053-1058, Dec 1972

[35] Schätz, B., Pfaller, C.: Integrating component tests to system tests. Electron. Notes Theor. Comput. Sci. 260, 225–241 (Jan 2010), http://dx.doi.org/10.1016/j.entcs.2009.12.040

[36] Sommerville, I.: Software Engineering. Addison-Wesley, Harlow, England, 9th edn. (2010)

[37] Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)

[38] Stocks, P., Carrington, D.: A Framework for Specification-Based Testing. IEEE Transactions on Software Engineering 22(11), 777–793 (Nov 1996)